Quantum Algorithms For String Problems

by

Ce Jin

B.Eng., Tsinghua University (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2022

Author . . . .
Department of Electrical Engineering and Computer Science
August 26, 2022

Certified by . . . .
Virginia Vassilevska Williams
Steven and Renee Finn Career Development Professor of
Electrical Engineering and Computer Science
Thesis Supervisor

Certified by . . . .
R. Ryan Williams
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Quantum Algorithms For String Problems

by

Ce Jin

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 2022, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

We design near-optimal quantum query algorithms for two important text processing problems: *Longest Common Substring* and *Lexicographically Minimal String Rotation*. Specifically, we show that:

- Longest Common Substring can be solved by a quantum algorithm in $\widetilde{O}(n^{2/3})$ time, improving upon the $\widetilde{O}(n^{5/6})$-time algorithm by Le Gall and Seddighin (2022). Moreover, given a length threshold $1 \le d \le n$, our algorithm decides in $n^{2/3+o(1)}/d^{1/6}$ time whether the longest common substring has length at least d, almost matching the $\Omega(n^{2/3}/d^{1/6})$ quantum query lower bound.

- Lexicographically Minimal String Rotation can be solved by a quantum algorithm in $n^{1/2+o(1)}$ time, improving upon the $\widetilde{O}(n^{3/4})$-time algorithm by Wang and Ying (2020), and almost matching the $\Omega(\sqrt{n})$ quantum query lower bound.

Our algorithm for Lexicographically Minimal String Rotation is obtained by speeding up a divide-and-conquer algorithm using nested Grover search and quantum minimum finding. Combining this divide-and-conquer idea with the deterministic sampling algorithm of Vishkin (1991) and Ramesh and Vinay (2003), we achieve a quantum speed-up of the *String Synchronizing Set* technique introduced by Kempa and Kociumaka (2019). Our algorithm for Longest Common Substring applies this string synchronizing set in the quantum walk framework.

Thesis Supervisor: Virginia Vassilevska Williams
Title: Steven and Renee Finn Career Development Professor of
Electrical Engineering and Computer Science

Thesis Supervisor: R. Ryan Williams
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my excellent advisors, Ryan Williams and Virginia Vassilevska Williams, for their continuing guidance in my research. Ryan and Virginia have always been supportive of me. They have introduced me to many fun topics in algorithms and complexity, provided me with various research opportunities, and also given me the freedom to explore my interest. When I got stuck on a problem, they often encouraged me to approach it from a different angle. I have benefited a lot from their advice.

Next, I would like to thank all my previous research mentors. Jelani Nelson hosted me at Harvard in summer 2018. This was my first successful research experience, which really boosted my confidence. In spring 2019, I visited Ryan Williams at MIT, and was also unofficially co-advised by Lijie Chen. Later that year, I participated in the ETH Zurich summer research program, hosted by Mohsen Ghaffari. I would also like to thank my undergraduate advisors from Tsinghua University, Ran Duan and Zhaohui Wei. More recently, I did a summer research internship at Google, Mountain View, hosted by Joshua R. Wang. I would not have made it this far without these mentors.

This thesis is based on two papers [5, 61]. I would like to thank my amazing collaborators on these two projects, Shyan Akmal and Jakob Nogler. I would also like to thank Shyan for organizing and inviting me to board games.

Finally, I would like to thank my parents for supporting me throughout my life and always encouraging me to explore math and computer science since I was a kid.

Contents

Chapter 1

Introduction

1.1 Background

The study of string processing algorithms is an important area of research in theoretical computer science, with applications in numerous fields including bioinformatics, data mining, plagiarism detection, etc. Many fundamental problems in this area have been known to have linear-time algorithms since over 40 years ago. Examples include *Exact String Matching* [68, 62], *Longest Common Substring* [93, 48, 19], and *(Lexicographically) Minimal String Rotation* [28, 86, 47]. These problems have also been studied extensively in the context of data structures, parallel algorithms, and low-space algorithms.

More recently, there has been growing interest in developing efficient *quantum algorithms* for these basic string problems. Given quantum query access to the input strings (defined in Section 2.3), it is sometimes possible to solve such problems in *sublinear* query complexity and time complexity. The earliest such result was given by Ramesh and Vinay [85], who combined Vishkin's deterministic sampling technique [91] with Grover search [55] to obtain a quantum algorithm for the Exact String Matching problem with near-optimal $\widetilde{O}(\sqrt{n})$ time complexity[1]. More recently, Le Gall and Seddighin [51] obtained sublinear-time quantum algorithms for various string problems, among them an $\widetilde{O}(n^{5/6})$-time algorithm for Longest Common Sub-

[1]We use $\widetilde{O}(\cdot), \widetilde{\Omega}(\cdot), \widetilde{\Theta}(\cdot)$ to hide $\operatorname{poly}\log n$ factors where n denotes the input length.

string (LCS) and an $\widetilde{O}(\sqrt{n})$-time algorithm for Longest Palindromic Substring (LPS). In developing these algorithms, they applied the quantum Exact String Matching algorithm [85] and Ambainis' Element Distinctness algorithm [9] as subroutines, and used periodicity arguments to reduce the number of candidate solutions to be checked. Another recent work by Wang and Ying [92] showed that Minimal String Rotation can be solved in $\widetilde{O}(n^{3/4})$ quantum time. Their algorithm was also based on quantum search primitives (including Grover search and quantum minimum finding [46]) and techniques borrowed from parallel string algorithms [16, 91, 57]. Other string problems previously studied in the quantum setting include Edit Distance [29, 34] and Regular Language Recognition [2, 10].

On the lower bound side, it has been shown that Longest Common Substring requires $\widetilde{\Omega}(n^{2/3})$ quantum query complexity (by a reduction [51] from the Element Distinctness problem [3, 73, 8]), and that Exact String Matching, Minimal String Rotation, and Longest Palindromic Substring all require $\Omega(\sqrt{n})$ quantum query complexity (by reductions [51, 92] from the unstructured search problem [24]). Le Gall and Seddighin [51] observed that although the classical algorithms for LCS and LPS are almost the same (both based on suffix trees [93]), the latter problem (with time complexity $\widetilde{\Theta}(\sqrt{n})$) is strictly easier than the former (with an $\widetilde{\Omega}(n^{2/3})$ lower bound) in the quantum query model.

Despite these results, our knowledge about the quantum computational complexities of basic string problems is far from complete. For the LCS problem and the Minimal String Rotation problem mentioned above, there are $n^{\Omega(1)}$ gaps between current upper bounds and lower bounds. Better upper bounds are only known in special cases: Le Gall and Seddighin [51] gave an $\widetilde{O}(n^{2/3})$-time algorithm for $(1-\varepsilon)$-approximating LCS in non-repetitive strings, matching the query lower bound in this setting. Wang and Ying [92] gave an $\widetilde{O}(\sqrt{n})$-time algorithm for Minimum String Rotation in randomly generated strings, and showed a matching average-case query lower bound. However, these algorithms do not immediately extend to the general cases.

1.2 Our Contribution

In this work, we develop new quantum query algorithms for Longest Common Substring and Minimal String Rotation, closing the gaps left open in previous works [51, 92]. Our algorithms are near-optimal and time-efficient: they have time complexities that match the corresponding query complexity lower bounds up to $n^{o(1)}$ factors.

1.2.1 Longest Common Substring

In the *Longest Common Substring (LCS)* problem, we are given two strings $S_1, S_2 \in \Sigma^n$, and want to compute $\mathrm{LCS}(S_1, S_2)$, defined as the maximum possible length d of their common substring $S_1[i \mathinner{.\,.} i+d-1] = S_2[j \mathinner{.\,.} j+d-1]$.[2] The previous quantum algorithm by Le Gall and Seddighin [51] actually solves the decisional problem, *LCS with threshold* d, which takes an extra parameter $1 \le d \le n$ and asks whether $\mathrm{LCS}(S_1, S_2) \ge d$ holds. Their algorithm for LCS with threshold d has time complexity $\widetilde{O}(\min\{n^{2/3} \cdot \sqrt{d}, n/\sqrt{d}\})$, which is $\widetilde{O}(n^{5/6})$ in the worst case $d = n^{1/3}$.

We give a new algorithm for LCS with threshold d, which outperforms the previous algorithm [51] for almost all $1 \ll d \ll n$.

Theorem 1.2.1 (LCS with threshold d, upper bound)**.** *Given $S_1, S_2 \in \Sigma^n$, there is a quantum algorithm that decides whether S_1, S_2 have a common substring of length d in $\widetilde{O}(n^{2/3}/d^{1/6-o(1)})$ quantum query complexity and time complexity.*

A schematic comparison of previous bound [51] and our new bound is in Fig. 1-1. In particular, we can compute $\mathrm{LCS}(S_1, S_2)$ in $\widetilde{O}(n^{2/3})$ quantum query and time complexity, by binary search over the threshold length d.

We also observe a quantum query lower bound that matches the upper bound up to $d^{o(1)}$ factors. Hence, we obtain an almost complete understanding of LCS with threshold d in the quantum query model.

[2] We remark that in the literature the same acronym could also refer to the Longest Common Subsequence problem. The difference is that a subsequence is not necessarily a contiguous part of the string. In this paper, we only consider the Longest Common Substring problem.

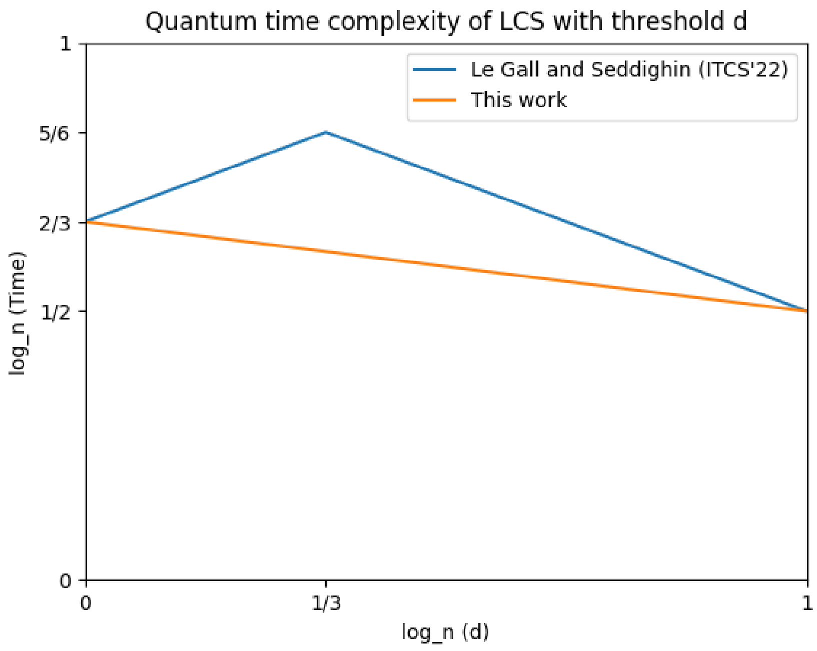

Figure 1-1: Quantum time complexity of LCS with threshold d

Theorem 1.2.2 (LCS with threshold d, lower bound)**.** *For $|\Sigma| \geq \Omega(n/d)$, deciding whether $S_1, S_2 \in \Sigma^n$ have a common substring of length d requires $\Omega(n^{2/3}/d^{1/6})$ quantum queries.*

The LCS with threshold d problem is particularly interesting from a quantum query complexity perspective, as its two extreme cases correspond to two fundamental problems in this field: the $d = 1$ case asks whether there exist i, j such that $S_1[i] = S_2[j]$, i.e., the (bipartite) Element distinctness problem, which has quantum query complexity $\Theta(n^{2/3})$ due to the celebrated results of Ambainis [9] and Aaronson and Shi [3]. The $d = n$ case asks to find i such that $S_1[i] \neq S_2[i]$, which is equivalent to the unstructured search problem, with well-known quantum query complexity $\Theta(n^{1/2})$ [24] achieved by Grover Search [55]. Theorem 1.2.1 and Theorem 1.2.2 indicate that the complexity of LCS with threshold d smoothly interpolates between the two extreme cases $d = 1$ and $d = n$ (up to subpolynomial factors).

1.2.2 Minimal String Rotation

In the *(Lexicographically) Minimal String Rotation* problem, we are given $S \in \Sigma^n$, and want to find a cyclic rotation of S with minimal lexicographical order. Two closely related problems are Minimal Suffix and Maximal Suffix. See Section 2.2 for formal definitions. These three problems have $\Omega(\sqrt{n})$ quantum query lower bound, proved by reduction from the unstructured search problem [92]. In this work, we obtain near-optimal quantum algorithms for these problems. In particular, we improve the previous $\widetilde{O}(n^{3/4})$-time algorithm by Wang and Ying [92] for Minimal String Rotation.

Theorem 1.2.3. *Minimal String Rotation, Maximal Suffix, and Minimal Suffix can be solved by a quantum algorithm with $n^{1/2+o(1)}$ query complexity and time complexity.*

1.2.3 Quantum String Synchronizing Sets

An important technical ingredient in our algorithm for LCS with threshold d (Theorem 1.2.1) is a quantum speed-up for constructing a *String Synchronizing Set*, a powerful tool for string algorithms recently introduced by Kempa and Kociumaka [63]. Informally speaking, for a length parameter $\tau \geq 1$, a τ-synchronizing set $\mathsf{A} \subseteq [1 \mathinner{.\,.} n]$ of a string $T \in \Sigma^n$ is a subset of synchronizing positions that are *consistently* sampled depending on their length-$\Theta(\tau)$ contexts in T, such that every non-periodic length-$\Theta(\tau)$ region in T is hit by at least one synchronizing position. A formal definition is given below.

Definition 1.2.4 (String synchronizing set [63])**.** For a string $T[1 \mathinner{.\,.} n]$ and a positive integer $1 \leq \tau \leq n/2$, we say $\mathsf{A} \subseteq [1 \mathinner{.\,.} n - 2\tau + 1]$ is a *τ-synchronizing set of T* if it satisfies the following properties:

- **Consistency:** If $T[i \mathinner{.\,.} i + 2\tau) = T[j \mathinner{.\,.} j + 2\tau)$, then $i \in \mathsf{A}$ if and only if $j \in \mathsf{A}$.

- **Density:** For $i \in [1 \mathinner{.\,.} n - 3\tau + 2]$, $\mathsf{A} \cap [i \mathinner{.\,.} i + \tau) = \emptyset$ if and only if $\mathsf{per}(T[i \mathinner{.\,.} i + 3\tau - 2]) \leq \tau/3$.

Kempa and Kociumaka [63] obtained a classical deterministic $O(n)$-time algorithm for constructing a τ-string synchronizing set of *optimal size* $\Theta(n/\tau)$, which was later applied in classical LCS algorithms [40] using an anchoring idea (see Section 1.3.1). However, in our sublinear-time quantum LCS application, we cannot afford the linear time of the classical construction algorithm in [63]. To overcome this issue, we design a quantum algorithm that provides efficient local access to elements in the synchronizing set A. The property of our construction is formally summarized in the following theorem.

Theorem 1.2.5 (Quantum string synchronizing set)**.** *Given string $T[1\,..\,n]$ and integer $1 \le \tau \le n/2$, there is a randomized $\mathsf{A} \subseteq [1\,..\,n-2\tau+1]$ generated from random seed $\sigma \in \{0,1\}^{\operatorname{poly}\log(n)}$, with the following properties:*

- ***Correctness:*** *A is always a τ-synchronizing set of T.*
- ***Sparsity:*** *For every $i \in [1\,..\,n-3\tau+2]$, $\mathbf{E}_\sigma\left[|\mathsf{A} \cap [i\,..\,i+\tau)|\right] \le \tau^{o(1)}$.*
- ***Efficient computability:*** *With high probability over σ, there is quantum algorithm that, given $i \in [1\,..\,n-3\tau+2]$, σ, and quantum query access to $T[i\,..\,i+3\tau-2]$, reports all the elements in $\mathsf{A} \cap [i\,..\,i+\tau)$ in*

  $$(\mathsf{cnt}+1) \cdot \tau^{\frac{1}{2}+o(1)} \cdot \operatorname{poly}\log(n)$$

  quantum time, where $\mathsf{cnt} = |\mathsf{A} \cap [i\,..\,i+\tau)|$ is the output count.

Compared with the previous classical construction [63], the expected size $|\mathsf{A}|$ in Theorem 1.2.5 is $(n/\tau) \cdot \tau^{o(1)}$, worse than optimal by a $\tau^{o(1)}$ factor. However, we can report each element in A in $\tau^{1/2+o(1)} \cdot \operatorname{poly}\log(n)$ quantum time, while in the classical construction each element takes $O(\tau)$ average time.

Theorem 1.2.5 plays a crucial role in our quantum LCS algorithm (Theorem 1.2.1). Given numerous known applications of string synchronizing sets in classical string algorithms [63, 6, 40, 64, 65, 66], we expect Theorem 1.2.5 to be a very useful tool in designing quantum string algorithms.

1.2.4 Bibliographic Note

The results and proofs presented in this thesis are extracted from the following two papers:

- *Near-optimal Quantum Algorithms for String Problems* joint with Shyan Akmal, which appeared in SODA 2022 [5], and
- *Quantum Speed-ups for String Synchronizing Sets, Longest Common Substring, and k-mismatch Matching*, an unpublished manuscript joint with Jakob Nogler [61].

Theorem 1.2.3 and a weaker version of Theorem 1.2.1 were proved in [5]. Theorem 1.2.1, Theorem 1.2.2, and Theorem 1.2.5 were proved in [61].

1.3 Technical Overview

We give high-level overviews of our quantum algorithms for Longest Common Substring, Minimal String Rotation, and String Synchronizing Sets.

1.3.1 Longest Common Substring

Le Gall and Seddighin [51, Section 3.1.1] observed a simple reduction from *LCS with threshold d* to the (bipartite version of) Element Distinctness problem, which asks whether the two input lists A, B contain a pair of identical items $A_i = B_j$: each item in the lists is a length-d substring of S_1 or S_2 (specified by the starting position), and the lexicographical order between two length-d substrings can be compared in $\mathcal{T} = \widetilde{O}(\sqrt{d})$ using binary search and Grover search (see Observation 2.5.1). Using Ambainis' [9] comparison-based algorithm in $\widetilde{O}(n^{2/3} \cdot \mathcal{T})$ time, LCS with threshold d can be solved in $\widetilde{O}(n^{2/3} \cdot \sqrt{d})$ time.

The anchoring technique. The inefficiency of the algorithm described above comes from the fact that there are $n - d + 1 = \Omega(n)$ positions to be considered in each input string. This seems rather unnecessary for larger d, since intuitively there is a lot of redundancy from the large overlap between these length-d substrings. This is the idea behind the so-called *anchoring* technique, which has been widely applied in designing classical algorithms for various versions of the LCS problem [88, 38, 13, 14, 23, 39, 40].

In this technique, we carefully pick subsets $\mathsf{C}_1, \mathsf{C}_2 \subseteq [n]$ of *anchors*, such that in a YES input instance there must exist an *anchored common substring*, i.e., a common string with occurrences $S_1[i_1 \mathinner{.\,.} i_2 + d) = S_2[i_2 \mathinner{.\,.} i_2 + d)$ and a shift $0 \le h < d$ such that $i_1 + h \in \mathsf{C}_1$ and $i_2 + h \in \mathsf{C}_2$. Then, the task reduces to the *Two String Families LCP problem* [38], where we want to find a pair of anchors $i_1' \in \mathsf{C}_1, i_2' \in \mathsf{C}_2$ that can be extended in both directions to get a length-d common substring, or equivalently, the longest common prefix of $S_1[i_1' \mathinner{.\,.}], S_2[i_2' \mathinner{.\,.}]$ and the longest common suffix of $S_1[\mathinner{.\,.} i_1' - 1], S_2[\mathinner{.\,.} i_2' - 1]$ have total length at least d.

To get better running time, we would like to *efficiently* construct anchor sets $\mathsf{C}_1, \mathsf{C}_2$ with *small size*. As observed in [40], such anchor sets can be constructed from τ-string synchronizing sets (Definition 1.2.4) introduced by Kempa and Kociumaka [63]: When $d > 3\tau$, a length-d common substring $S_1[i_1 \mathinner{.\,.} i_1 + d - 1] = S_2[i_2 \mathinner{.\,.} i_2 + d - 1]$ imply that these two regions contain consistently sampled synchronizing positions, which means that they have common length-d substrings anchored at some synchronizing positions in this region (this argument ignores the highly-periodic case, which can be dealt with otherwise). Kempa and Kociumaka's construction leads to anchor sets with optimal $O(n/d)$ size, but we cannot afford to use their linear-time construction algorithm in our sublinear-time quantum setting. Instead, we use the version of string synchronizing sets with a quantum speed-up (Theorem 1.2.5), and construct an anchor set of size $n/d^{1-o(1)}$, in which each element can be reported in $\widetilde{O}(d^{1/2+o(1)})$ quantum time (Theorem 3.1.2).

Anchoring via quantum walks. Now we explain how to use small and explicit anchor sets to obtain better quantum LCS algorithms with time complexity $\widetilde{O}(m^{2/3} \cdot (\sqrt{d} + \mathcal{T})) = \widetilde{O}(n^{2/3}/d^{1/6-o(1)})$ (Theorem 3.1.4), where $m = n/d^{1/2-o(1)}$ is the number of anchors, and $\mathcal{T} = \widetilde{O}(d^{1/2+o(1)})$ is the time complexity of computing the i^{th} anchor.

Our algorithm uses the *MNRS quantum walk* framework [76] (see Section 2.6) on Johnson graphs. Informally speaking, to apply this framework, we need to solve the following dynamic problem: maintain a *subset* of r anchors which undergoes insertions and deletions (called *update steps*), and in each query (called a *checking step*) we need to solve the Two String Families LCP problem on this subset, i.e., answer whether the current subset contains a pair of anchors that can extend to a length-d common substring. If each update step takes time $\mathcal{U}$, and each checking step takes time $\mathcal{C}$, then the MNRS quantum walk algorithm has overall running time $\widetilde{O}(r\,\mathcal{U}+\frac{m}{r}\,(\sqrt{r}\,\mathcal{U}+\mathcal{C}))$. We will achieve $\mathcal{U} = \widetilde{O}(\sqrt{d}+T)$ and $\mathcal{C} = \widetilde{O}(\sqrt{rd})$, and obtain the claimed time complexity by setting $r = m^{2/3}$.

To solve this dynamic problem, we maintain the lexicographical ordering of the length-d substrings specified by the current subset of anchors, as well as the corresponding LCP array which contains the length of the longest common prefix between every two lexicographically adjacent substrings. Note that the maintained information uniquely defines the *compact trie* of these substrings. This information can be updated easily after each insertion (or deletion) operation: we first compute the inserted anchor in T time, and then use binary search with Grover search to find its lexicographical rank and the LCP values with its neighbors, in $\widetilde{O}(\sqrt{d})$ quantum time.

The maintained information will be useful for the checking step. In fact, if we only care about query complexity, then we are already done, since the maintained information already uniquely determines the answer of the Two String Families LCP problem, and no additional queries to the input strings are needed. The main challenge is to implement this checking step time-efficiently. Unfortunately, the classical near-linear-time algorithm [38] for solving the Two String Families LCP problem is too slow compared to our goal of $\mathcal{C} = \widetilde{O}(\sqrt{rd})$, and it is not clear how to obtain a quantum speedup over this classical algorithm. Hence, we should try to dynamically maintain the solution using data structures, instead of solving it from scratch every time. In fact, such a data structure with $\operatorname{poly}\log(n)$ time per operation was already given by Charalampopoulos, Gawrychowski, and Pokorski [39], and was used to obtain a classical data structure for maintaining Longest Common Substring under character substitutions. However, this data structure cannot be applied to the quantum walk algorithm, since it violates two requirements that are crucial for the correctness of quantum walk algorithms: (1) It should have worst-case time complexity (instead of being amortized), and (2) it should be *history-independent* (see the discussion in Section 3.3.1 for more details). Instead, we will design a different data structure that satisfies these two requirements, and can solve the Two String Families LCP problem on the maintained subset in $\widetilde{O}(\sqrt{rd})$ quantum time. This time complexity is worse than the $\operatorname{poly}\log(n)$ time achieved by the classical data structure of [39], but suffices for our application.

A technical hurdle: limitations of 2D range query data structures. Our solution for the Two String Families LCP problem is straightforward, but a key component in the algorithm relies on *dynamic 2-dimensional orthogonal range queries*. This is a well-studied problem in the data structure literature, and many $\operatorname{poly}\log n$-time data structures are known (see [94, 80, 37] and the references therein). However, for our results, the 2-dimensional (2D) range query data structure in question has to satisfy not only the two requirements mentioned above, but also a third requirement of being *comparison-based*. In particular, we are not allowed to treat the coordinates of the 2D points as $\operatorname{poly}(n)$-bounded integers, because the coordinates actually correspond to substrings of the input string, and should be compared by lexicographical order. Unfortunately, no data structures satisfying all three requirements are known.

To bypass this difficulty, our novel idea is to use a sampling procedure that lets us estimate the rank of a coordinate of the inserted 2D point among all the possible coordinates, which effectively allows us to convert the non-integer coordinates into integer coordinates. By a version of the Balls-and-Bins hashing argument, the inaccuracy incurred by the sampling can be controlled for *most* of the vertices on the Johnson graph which the quantum walk operates on. This then lets us apply 2D range query data structures over integer coordinates (see Section 3.3.3 for the details of this argument), which can be implemented with worst-case time complexity and history-independence as required. Combining this method with the tools and ideas mentioned before lets us get a time-efficient implementation of the quantum walk algorithm for computing the LCS.

1.3.2 Minimal String Rotation

In the Minimal String Rotation problem, we are given a string S of length n and are tasked with finding the cyclic rotation of S which is lexicographically the smallest. We sketch the main ideas of our improved quantum algorithm for Minimal String Rotation by comparing it to the previous best solution for this problem.

The simplest version of Wang and Ying's algorithm [92, Theorem 5.2] works by identifying a small prefix of the minimal rotation using Grover search, and then applying pattern matching with this small prefix to find the starting position of the minimum rotation. More concretely, let B be some size parameter. By quantum minimum finding over all prefixes of length B among the rotations of S, we can find the length-B prefix P of the minimal rotation in asymptotically $\sqrt{B} \cdot \sqrt{n}$ time. Next, split the string S into $\Theta(n/B)$ blocks of size $\Theta(B)$ each. Within each block, we find the *leftmost* occurrence of P via quantum Exact String Matching [85]. It turns out that one of these positions is guaranteed to be a starting position of the minimal rotation (this property is called an "exclusion rule" or "Ricochet Property" in the literature). By minimum finding over these $O(n/B)$ candidate starting positions (and comparisons of length-n strings via Grover search), we can find the true minimum rotation in asymptotically $\sqrt{n/B} \cdot \sqrt{n}$ time. So overall the algorithm takes asymptotically

$$\sqrt{Bn} + (n/\sqrt{B})$$

time, which is minimized at $B = \sqrt{n}$ and yields a runtime of $\widetilde{O}(n^{3/4})$.

This algorithm is inefficient in its first step, where it uses quantum minimum finding to obtain the minimum length-B prefix P. The length-B prefixes we are searching over all come from rotations of the same string S. Due to this common structure, we should be able to find their minimum more efficiently than just using the generic algorithm for minimum finding. At a high level, we improve this step by finding P using *recursion* instead. Intuitively, this is possible because the Minimal Rotation problem is already about finding the minimum "prefix" (just of length n) among rotations of S. This then yields a recursive algorithm running in $n^{1/2+o(1)}$ quantum time. This recursion yields a tournament-tree-like structure, on which a classical divide-and-conquer algorithm would find the "winner" (namely, the minimal rotation) in $O(n \log n)$ time, while a quantum algorithm can achieve an almost-quadratic speed-up.

In the presentation of this algorithm in Chapter 4, we use a chain of reductions and actually solve a more general problem to get this recursion to work. The argument also relies on a new "exclusion rule," adapted from previous work, to prove that we only need to consider a constant number of candidate starting positions of the minimum rotation within each small block of the input string.

1.3.3 Quantum String Synchronizing Sets

We outline our construction of τ-synchronizing sets with $\widetilde{O}(\tau^{1/2+o(1)})$ quantum reporting time per element (Theorem 1.2.5). For simplicity, here we only consider the non-periodic case, i.e., we assume that the input string $T[1\mathinner{.\,.} n]$ does not contain any length-τ substring with period at most $\tau/3$.

In this non-periodic case, our starting point is a simple randomized construction of [63], which is much simpler than their deterministic construction (whose sequential nature makes it more difficult to have a quantum speed-up). Their idea is to *pick local minimizers of a random hash function.* More specifically, sample a random hash value $\phi(S)$ for every $S \in \Sigma^\tau$, and denote $\Phi(i) := \phi(T[i\mathinner{.\,.} i+\tau))$. Then, include i in the synchronizing set if and only if $\min_{j\in[i\mathinner{.\,.} i+\tau]} \Phi(j)$ is achieved at $j \in \{i, i+\tau\}$. It is straightforward to verify that, (1) whether $i \in \mathsf{A}$ is completely determined by $T[i\mathinner{.\,.} i+2\tau)$ and the randomness, and (2) every length-τ interval contains at least one $i \in \mathsf{A}$. So A is indeed an τ-synchronizing set. Then, the non-periodic assumption ensures that nearby length-τ substrings of T are distinct, so that the probability of $i \in \mathsf{A}$ is $1/\Omega(\tau)$ for every i, which implies the sparsity of A in expectation.

To implement the above idea in the quantum setting, the first challenge is to implement a hash function ϕ on length-τ substrings that can be evaluated in at most

$\widetilde{O}(\tau^{1/2+o(1)})$ quantum time. Naturally, the hash function should be random enough. A minimal (but not sufficient) requirement seems to be that ϕ should at least be able to distinguish two different strings $x, y \in \Sigma^\tau$ by outputting different values $\phi(x) \neq \phi(y)$ with good probability. However, it is not clear how such hash family can be implemented in only $\widetilde{O}(\tau^{1/2+o(1)})$ quantum time. Many standard hash functions that have this property, such as Karp-Rabin fingerprints, provably require at least $\Omega(\tau)$ query complexity.

To overcome this challenge, we observe that it is not necessary to use a hash family with full distinguishing ability. In order for the randomized construction to work, we only need that the hash values of two heavily overlapping length-τ substrings to behave like random. Fortunately, this weaker requirement can be satisfied using the *deterministic sampling* method of Vishkin [91]. This technique was originally used for parallel algorithms for exact string matching [91], and was later adapted into a quantum algorithm for exact string matching in $\widetilde{O}(\sqrt{n})$ time by Ramesh and Vinay [85], as well as some other types of exact string matching problem, e.g., [52, 42]. In the context of string matching, the idea is to carefully sample $O(\log n)$ positions in the pattern, so that a candidate match in the text that agrees on all the sampled positions can be used to rule out other nearby candidate matches, and hence save computation by reducing the number of candidate matches that have to undergo a linear-time full check against the pattern. In our situation, we use the quantum algorithm for deterministic sampling [85] in $\widetilde{O}(\sqrt{\tau})$ time, and use these carefully sampled positions to build a hash function that is guaranteed to evaluate to different values on two heavily overlapping strings. To the best of our knowledge, this is the first application of deterministic sampling in a completely different context than designing string matching algorithms.

Having designed a suitable hash function ϕ, a straightforward attempt to report a synchronizing position is to use the quantum minimum finding algorithm [46] on a length-τ interval $[j \mathinner{.\,.} j+\tau)$ to find the $i \in [j \mathinner{.\,.} j+\tau)$ with minimal hash value $\Phi(i) = \phi(T[i \mathinner{.\,.} i+\tau))$. This incurs $O(\sqrt{\tau})$ evaluations of the hash function, each taking $\widetilde{O}(\sqrt{\tau})$ quantum time, which would still be $\widetilde{O}(\tau)$ in total, slower than our goal of $\widetilde{O}(\tau^{1/2+o(1)})$.

To obtain better quantum query and time complexity, we borrow the recursion idea from our Minimal String Rotation algorithm described earlier. We further modify our construction of the hash function ϕ, so that one can find the minimal hash value in a length-τ interval in a tournament tree-like fashion: The leaves are the τ candidates, each internal node picks the minimum hash value among its children using quantum minimum finding, and the root will be the minimum among all τ intervals. Here, the crucial point is to make sure that comparing two nodes in a lower level (corresponding to two closer candidates) can take less time. This makes sure that the total quantum time complexity of finding the minimum hash value is still $\tau^{1/2+o(1)}$. Compared to the recursive algorithm for Minimal String Rotation, the construction here requires more technical details to deal with the case where a node in the tournament tree returns multiple minimizers.

1.4 Related Work

Quantum algorithms on string problems. Wang and Ying [92] improved the logarithmic factors of the quantum Exact String Matching algorithm by Ramesh and Vinay [85] (and filled in several gaps in their original proof).

Another important string problem is computing the *edit distance* between two strings (the minimum number of deletions, insertions, and substitutions needed to turn one string into the other). The best known classical algorithm has $O(n^2/\log^2 n)$ time complexity [78], which is near-optimal under the Strong Exponential Time Hypothesis [20]. It is open whether quantum algorithms can compute edit distance in truly subquadratic time. For the approximate version of the edit distance problem, the breakthrough work of Boroujeni et al. [29] gave a truly subquadratic time quantum algorithm for computing a constant factor approximation. The quantum subroutines of this algorithm were subsequently replaced with classical randomized algorithms in [36] to get a truly subquadratic classical algorithm that approximates the edit distance to a constant factor.

Le Gall and Seddighin [51] also considered the $(1+\varepsilon)$-approximate Ulam distance problem (i.e., edit distance on non-repetitive strings), and showed a quantum algorithm with near-optimal $\widetilde{O}(\sqrt{n})$ time complexity. Their algorithm was based on the classical algorithm by Naumovitz, Saks, and Seshadhri [81].

Montarano [79] gave quantum algorithms for the d-dimensional pattern matching problem with random inputs. Ambainis et al. [10] gave quantum algorithms for deciding Dyck languages. There are also some results [12, 41] on string problems with non-standard quantum queries to the input.

Quantum walks and time-efficient quantum algorithms. Quantum walks [89, 9, 76] are a useful method to obtain query-efficient quantum algorithms for many important problems, such as Element Distinctness [9] and Triangle Finding [77, 60, 74]. Ambainis showed that the query-efficient algorithm for element distinctness [9] can also be implemented in a time-efficient manner with only a $\operatorname{poly}\log(n)$ blowup, by applying history-independent data structures in the quantum walk. Since then, this "quantum walk plus data structure" strategy has been used in many quantum algorithms to obtain improved time complexity. For example, Belovs, Childs, Jeffery, Kothari, and Magniez [22] used nested quantum walk with Ambainis' data structure to obtain time-efficient algorithms for the 3-distinctness problem. Bernstein, Jeffery, Lange, and Meurer [25] designed a simpler data structure called quantum radix tree [59], and applied it in their quantum walk algorithms for the Subset Sum problem on random input. Aaronson, Chia, Lin, Wang, and Zhang [1] gave a quantum walk algorithm for the Closest-Pair problem in $O(1)$-dimensional space with near-optimal time complexity $\widetilde{O}(n^{2/3})$. The previous $\widetilde{O}(n^{2/3})$-time algorithm for approximating LCS in non-repetitive strings [51] also applied quantum walks.

On the other hand, query-efficient quantum algorithms do not always have time-efficient implementations. This motivated the study of *quantum fine-grained complexity*. Aaronson et al. [1] formulated the QSETH conjecture, which is a quantum analog of the classical Strong Exponential Time Hypothesis, and showed that Orthogonal Vectors and Closest-Pair in $\operatorname{poly}\log(n)$-dimensional space require $n^{1-o(1)}$ quantum time under QSETH. In contrast, these two problems have simple quantum walk algorithms with only $O(n^{2/3})$ query complexity. Buhrman, Patro, and Speelman [34] formulated another version of QSETH, which implies a conditional $\Omega(n^{1.5})$-time lower bound for quantum algorithms solving the edit distance problem. Recently, Buhrman, Loff, Patro, and Speelman [33] proposed the quantum 3SUM hypothesis, and used it to show that the quadratic quantum speedups obtained by Ambainis and Larka [11] for many computational geometry problems are conditionally optimal. Notably, in their fine-grained reductions, they employed a quantum walk with data structures to bypass the linear-time preprocessing stage that a naive approach would require.

Classical string algorithms. We refer readers to several excellent textbooks [56, 44, 43] on string algorithms.

Weiner [93] introduced the suffix tree and gave a linear-time algorithm for computing the LCS of two strings over a constant-sized alphabet. For polynomially-bounded integer alphabets, Farach's construction of suffix trees [48] implies an linear-time algorithm for LCS. Babenko and Starikovskaya [19] gave an algorithm for LCS based on suffix arrays. Recently, Charalampopoulos, Kociumaka, Pissis, and Radoszewski [40] gave faster word-RAM algorithms for LCS on compactly represented input strings over a small alphabet. The LCS problem has also been studied in the settings of time-space tradeoffs [88, 72, 23], approximate matching [19, 4, 50, 90, 87, 71, 38, 54], and dynamic data structures [13, 14, 39].

Booth [28] and Shiloach [86] gave the first linear time algorithms for the Minimal String Rotation problem. Later, Duval [47] gave a constant-space linear-time algorithm for computing the *Lyndon factorization* of a string, which can be used to compute the minimal rotation, maximal suffix, and minimal suffix. Duval's algorithm can also compute the minimal suffix and maximal suffix for every prefix of the input string. Apostolico and Crochemore [15] gave a linear-time algorithm for computing the minimal rotation of every prefix of the input string. Parallel algorithms for Minimal String Rotation were given by Iliopoulos and Smyth [57]. There are data structures [18, 17, 69] that, given a substring specified by its position and length in the input string, can efficiently answer its minimal suffix, maximal suffix, and minimal rotation.

1.5 Organization

In Chapter 2 we introduce basic definitions and useful lemmas that will be used throughout the paper. In Chapter 3, we present our algorithm for *Longest Common Substring*. In Chapter 4, we present our algorithm for *Minimal String Rotation* and several related problems. In Chapter 5 we present our construction of string synchronizing sets with quantum speedup. Our LCS algorithm in Chapter 3 uses the main theorems proved in Chapters 4 and 5 as black boxes. Finally, we mention several open problems in Chapter 6.

Chapter 2

Preliminaries

2.1 Notations and Basic Properties of Strings

We use $\widetilde{O}(\cdot), \widetilde{\Omega}(\cdot), \widetilde{\Theta}(\cdot)$ to hide $\operatorname{poly}\log(n)$ factors, where n is the input size. In particular, $\widetilde{O}(1)$ means $O(\operatorname{poly}\log n)$.

Define sets $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ and $\mathbb{N}^+ = \{1, 2, 3, \dots\}$. For every positive integer n, let $[n] = \{1, 2, \dots, n\}$. For integers $i \le j$, let $[i \,..\, j] = \{i, i+1, \dots, j\}$ denote the set of integers in the closed interval $[i, j]$. We define $[i \,..\, j), (i \,..\, j]$, and $(i \,..\, j)$ analogously.

We consider strings over a *polynomially-bounded integer alphabet* $\Sigma = [1 \,..\, n^{O(1)}]$. A string $S \in \Sigma^n$ is a sequence of characters $S = S[1]S[2]\cdots S[n]$ from the alphabet Σ (we use 1-based indexing). The *concatenation* of two strings $S, T \in \Sigma^*$ is denoted by ST. The *reversed string* of S is denoted by $S^R = S[n]S[n-1]\cdots S[1]$.

Given a string S of length $|S| = n$, a *substring* of S is any string of the form $S[i \,..\, j] = S[i]S[i+1]\cdots S[j]$ for some indices $1 \le i \le j \le n$. We sometimes use $S[i \,..\, j) = S[i]S[i+1]\cdots S[j-1]$ and $S(i \,..\, j] = S[i+1]\cdots S[j-1]S[j]$ to denote substrings. A substring $S[1 \,..\, j]$ is called a *prefix* of S, and a substring $S[i \,..\, n]$ is called a *suffix* of S. For two strings S, T, let $\mathsf{lcp}(S, T) = \max\{j : j \le \min\{|S|, |T|\}, S[1 \,..\, j] = T[1 \,..\, j]\}$ denote the length of their *longest common prefix*.

We say string S is *lexicographically smaller* than string T (denoted $S \prec T$) if either S is a proper prefix of T (i.e., $|S| < |T|$ and $S = T[1 \,..\, |S|]$), or $\ell = \mathsf{lcp}(S, T) <$

$\min\{|S|, |T|\}$ and $S[\ell+1] < T[\ell+1]$. The notations $\succ, \preceq, \succeq$ are defined analogously. The following easy-to-prove and well-known fact has been widely used in string data structures and algorithms.

Lemma 2.1.1 (e.g. [67, Lemma 1]). *Given strings $S_1 \preceq S_2 \preceq \cdots \preceq S_m$, we have $\mathsf{lcp}(S_1, S_m) = \min_{1 \le i < m}\{\mathsf{lcp}(S_i, S_{i+1})\}$.*

For a positive integer $p \le |S|$, we say p is a *period* of S if $S[i] = S[i+p]$ holds for all $1 \le i \le |S| - p$. We refer to the minimal period of S as *the period* of S, and denote it by $\mathsf{per}(S)$. If $\mathsf{per}(S) \le |S|/2$, we say that S is *periodic*. A *run* in T is a periodic substring that cannot be extended (to the left nor to the right) without an increase of its shortest period. We need the following well-known lemmas about periodicity in strings.

Lemma 2.1.2 (Weak periodicity lemma, [49]). *If a string S has periods p and q such that $p + q \le |S|$, then $\gcd(p, q)$ is also a period of S.*

Lemma 2.1.3 (Structure of substring occurrences, e.g., [82, 70]). *Let S, T be two strings with $2|T|/3 \le |S| \le |T|$, and let $T[k_1 \mathinner{.\,.} k_1 + |S|) = T[k_2 \mathinner{.\,.} k_2 + |S|) = \cdots = T[k_d \mathinner{.\,.} k_d + |S|) = S$ be all the occurrences of S in T (where $k_j < k_{j+1}$ for $1 \le j < d$). Then, $k_1, k_2, \ldots, k_d$ form an arithmetic progression. Moreover, if $d \ge 2$, then $\mathsf{per}(S) = k_2 - k_1$.*

We say string S is a *(cyclic) rotation* of string T, if $|S| = |T| = n$ and there exists an index $1 \le i \le n$ such that $S = T[i \mathinner{.\,.} n]T[1 \mathinner{.\,.} i-1]$. For a periodic string S with shortest period $\mathsf{per}(S) = p$, the *Lyndon root* of S is defined as the lexicographically minimal rotation of $S[1 \mathinner{.\,.} p]$.

2.2 Problem Definitions

We give formal definitions of the string problems considered in this paper.

Longest Common Substring (LCS)

Input: Two strings S_1, S_2

Task: Output the maximum length ℓ such that $S_1[i_1 \mathinner{..} i_1 + \ell) = S_2[i_2 \mathinner{..} i_2 + \ell)$ for some $i_1 \in [|S_1| - \ell + 1], i_2 \in [|S_2| - \ell + 1]$.

The length ℓ of the longest common substring of S_1, S_2 is denoted as $\mathrm{LCS}(S_1, S_2)$. We only require the algorithm to output the length ℓ; the locations i_1, i_2 can be found by a binary search. We also consider the following decisional problem.

LCS with Threshold d

Input: Two strings S_1, S_2 and a threshold parameter $d \geq 1$

Task: Decide whether $\mathrm{LCS}(S_1, S_2) \geq d$.

The following three problems involve the lexicographical order of substrings.

(Lexicographically) Minimal String Rotation

Input: A string S

Task: Output a position $i \in [1 \mathinner{..} |S|]$ such that $S[i \mathinner{..} |S|]S[1 \mathinner{..} i-1] \preceq S[j \mathinner{..} |S|]S[1 \mathinner{..} j-1]$ holds for all $j \in [1 \mathinner{..} |S|]$. If there are multiple solutions, output the smallest such i.

Maximal Suffix

Input: A string S

Task: Output the position $i \in [1 \mathinner{..} |S|]$ such that $S[i \mathinner{..} |S|] \succ S[j \mathinner{..} |S|]$ holds for all $j \in [|S|] \setminus \{i\}$.

Minimal Suffix

Input: A string S

Task: Output the position $i \in [1 \mathinner{..} |S|]$ such that $S[i \mathinner{..} |S|] \prec S[j \mathinner{..} |S|]$ holds for all $j \in [|S|] \setminus \{i\}$.

The following problem is the most basic problem in string algorithms.

Exact String Matching

Input: Two strings T, P with $|T| \geq |P|$

Task: Output the minimum position i such that $T[i \mathinner{..} i + |P|) = P$.

2.3 Computational Model

We assume the input strings can be accessed in a quantum query model [7, 32], which is standard in the literature of quantum algorithms. More precisely, letting S be an input string of length n, we have access to an oracle O_S that, for any index $i \in [n]$ and any $b \in \Sigma$, performs the unitary mapping $O_S \colon |i, b\rangle \mapsto |i, b \oplus S[i]\rangle$, where $\oplus$ denotes the XOR operation on the binary encodings of characters. The oracles can be queried in superposition, and each query has unit cost. Besides the input queries, the algorithm can also apply intermediate unitary operators that are independent of the input oracles. Finally, the query algorithm should return the correct answer with success probability at least $2/3$ (which can be boosted to high probability by a majority vote over $O(\log n)$ repetitions).[1] The *query complexity* of an algorithm is the number of queries it makes to the input oracles.

In this paper, we are also interested in the *time complexity* of the quantum algorithms, which counts not only the queries to the input oracles, but also the elementary gates [21] for implementing the unitary operators that are independent of the input. Similar to previous works (e.g., [9, 1, 51]), in order to implement the query algorithms in a time-efficient manner, we also need the *quantum random access gate*, defined as

$$|i, b, z_1, \dots, z_m\rangle \mapsto |i, z_i, z_1, \dots, z_{i-1}, b, z_{i+1}, \dots, z_m\rangle,$$

to access at unit cost the i^{th} element from the quantum working memory $|z_1, \dots, z_m\rangle$. Assuming quantum random access, a classical time-$\mathcal{T}$ algorithm that uses random access memory can be converted into a quantum subroutine in time $O(\mathcal{T})$, which can be invoked by quantum search primitives such as Grover search.

In this thesis we do not seek to optimize the log-factors in the query complexity or the time complexity of our algorithms.

[1]We say an algorithm succeeds *with high probability (w.h.p)*, if the success probability can be made at least $1 - 1/n^c$ for any desired constant $c > 1$.

2.4 Basic Quantum Primitives

We use the following basic quantum algorithms.

Grover search (Amplitude amplification) [55, 31]. Let $f\colon [n] \to \{0,1\}$ be a function, where $f(i)$ for each $i \in [n]$ can be evaluated in time $\mathcal{T}$. There is a quantum algorithm that, with high probability, finds an $x \in f^{-1}(1)$ or report that $f^{-1}(1)$ is empty, in $\widetilde{O}(\sqrt{n} \cdot \mathcal{T})$ time. Moreover, if it is guaranteed that either $|f^{-1}(1)| \geq M$ or $|f^{-1}(1)| = 0$ holds, then the algorithm runs in $\widetilde{O}(\sqrt{n/M} \cdot \mathcal{T})$ time.

Quantum minimum finding [46]. Let $x_1, \ldots, x_n$ be n items with a total order, where each pair of x_i and x_j can be compared in time $\mathcal{T}$. There is a quantum algorithm that, with high probability, finds the minimum item among $x_1, \ldots, x_n$ in $\widetilde{O}(\sqrt{n} \cdot \mathcal{T})$ time.

2.5 Quantum Algorithms on Strings

We review some known quantum algorithms on strings. The following algorithm follows from a simple binary search composed with Grover search.

Observation 2.5.1 (Finding longest common prefix)**.** *Given $S, T \in \Sigma^n$, there is an $\widetilde{O}(\sqrt{n})$-time quantum algorithm that computes $\mathsf{lcp}(S,T)$, and decides whether $S \preceq T$.*

Proof. Note that we can use Grover search to decide whether two strings are identical in $\widetilde{O}(\sqrt{n})$ time. Then we can compute $\mathsf{lcp}(S,T)$ by a simple binary search over the length of the prefix. After that we can easily compare their lexicographical order by comparing the next position. $\square$

Ramesh and Vinay [85] combined Grover search with the deterministic sampling technique of Vishkin [91], and obtained a near-optimal quantum algorithm for the Exact String Matching problem.

Theorem 2.5.2 ([85]). *Given pattern $P \in \Sigma^m$ and text $T \in \Sigma^n$ with $n \geq m$, there is an $\widetilde{O}(\sqrt{n})$-time quantum algorithm that finds an occurrence of P in T (or reports none exists).*

Kociumaka, Radoszewski, Rytter, and Waleń [70] showed that computing $\mathsf{per}(S)$ can be reduced to $O(\log |S|)$ instances of exact pattern matching and longest common prefix involving substrings of S. Hence, we have the following corollary.

Corollary 2.5.3 (Finding period). *Given $S \in \Sigma^n$, there is a quantum algorithm in $\widetilde{O}(\sqrt{n})$ time that computes $\mathsf{per}(S)$.*

2.6 Quantum Walks

We use the quantum walk framework [9, 89] developed by Magniez, Nayak, Roland, and Santha [76], and apply it on Johnson graphs.

The Johnson graph $J(m, r)$ has $\binom{m}{r}$ vertices, each being a subset of $[m]$ with size r, where two vertices in the graph $A, B \in \binom{[m]}{r}$ are connected by an edge if and only if $|A \cap B| = r - 1$, or equivalently there exist $a \in A, b \in [m] \setminus A$ such that $B = (A \setminus \{a\}) \cup \{b\}$. Depending on the application, we usually identify a special subset of the vertices $V_{\mathsf{marked}} \subseteq \binom{[m]}{r}$ as being *marked*. The quantum walk is analogous to a random walk on the Johnson graph attempting to find a marked vertex, but provides quantum speed-up compared to the classical random walk. The vertices in the Johnson graph are also called the states of the walk.

In the quantum walk algorithm, each vertex $K \in \binom{[m]}{r}$ is associated with a data structure $D(K)$. The setup cost $\mathcal{S}$ is the cost to set up the data structure $D(K)$ for any $K \in \binom{[m]}{r}$, where the cost could be measured in query complexity or time complexity. The checking cost $\mathcal{C}$ is the cost to check whether K is a marked vertex, given the data structure $D(K)$. The update cost $\mathcal{U}$ is the cost of updating the data structure from $D(K)$ to $D(K')$, where $K' = (K \setminus \{a\}) \cup \{b\}$ is an adjacent vertex specified by $a \in K, b \in [m] \setminus K$. The MNRS quantum walk algorithm can be summarized as follows.

Theorem 2.6.1 (MNRS quantum walk [76]). *Suppose $|V_{\mathsf{marked}}|/\binom{m}{r} \geq \varepsilon$ whenever V_{marked} is non-empty. Then there is a quantum algorithm that with high probability determines if V_{marked} is empty or finds a marked vertex, with cost of order $\mathcal{S} + \frac{1}{\sqrt{\varepsilon}}(\sqrt{r} \cdot \mathcal{U} + \mathcal{C})$.*

Readers unfamiliar with the quantum walk approach are referred to [45, Section 8.3.2] for a quick application of this theorem to solve the Element Distinctness problem using $O(n^{2/3})$ quantum queries. This algorithm can be implemented in $\widetilde{O}(n^{2/3})$ time by carefully designing the data structures to support time-efficient insertion, deletion, and searching [9, Section 6.2]. We elaborate on the issue of time efficiency when we apply quantum walks in our algorithm in Section 3.3.

2.7 Pseudorandomness

We use the following min-wise independent hash family.

Lemma 2.7.1 (Approximate min-wise independent hash family, follows from [58]). *Given integer parameters $n \geq 1$ and $N \geq c \cdot n^3$ (for some constant c), there is a hash family $\mathcal{H} = \{h\colon [N] \to [N^2]\}$, where each $h \in \mathcal{H}$ is an* injective *function that can be specified using $O(\log N)$ bits, and can be evaluated at any point in* $\operatorname{poly}\log(N)$ *time.*[2]

Moreover, $\mathcal{H}$ satisfies approximate min-wise independence*: for any $x \in [N]$ and subset $X \subseteq [N] \setminus \{x\}$ of size $|X| \leq n$,*

$$\mathop{\mathbf{Pr}}_{h \in \mathcal{H}}\left[h(x) < \min\{h(x') : x' \in X\}\right] \in \frac{1}{|X|+1} \cdot (1 \pm 0.1).$$

[2]The original definition in [58] used functions from $[N] \to [N]$ and does not guarantee injectivity with probability 1. Here we can guarantee injectivity by simply attaching the input string to the output. Doing this will slightly simplify some of the presentation later.

Chapter 3

Longest Common Substring

In this chapter, we study the Longest Common Substring problem. Our upper bound (Theorem 1.2.1) is proved in Sections 3.1 to 3.4, and a matching lower bound (Theorem 1.2.2) is proved in Section 3.5. Our upper bound proof uses Theorem 1.2.3 (proved in Chapter 4) and Theorem 1.2.5 (proved in Chapter 5) as black boxes.

3.1 The Anchoring Technique

Consider the decision version of LCS with length threshold $d \geq 1$, where we want to decide whether two input strings $S_1, S_2 \in \Sigma^*$ have a length-d common substring $S_1[i_1 \mathinner{.\,.} i_1 + d) = S_2[i_2 \mathinner{.\,.} i_2 + d)$ for some $i_1 \in [|S_1| - d + 1], i_2 \in [|S_2| - d + 1]$. The algorithm for computing $\mathrm{LCS}(S_1, S_2)$ then follows from a binary search over the threshold d. We assume $d \geq 100$ to avoid corner cases in later analysis; for smaller d, the problem can be solved in $\widetilde{O}(n^{2/3}d^{1/2}) = \widetilde{O}(n^{2/3})$ time by reducing to the (bipartite version of) Element Distinctness problem [51, Section 3.1.1] and applying Ambainis' algorithm [9].

To simplify the presentation, we concatenate the two input strings S_1, S_2 into $T := S_1\$S_2$, where \$ is a delimiter symbol that does not appear in the input strings, and let $n = |T| = |S_1| + 1 + |S_2|$. So $S_1[i] = T[i]$ for all $i \in [1 \mathinner{.\,.} |S_1|]$, and $S_2[j] = T[|S_1| + 1 + j]$ for all $j \in [1 \mathinner{.\,.} |S_2|]$.

Our algorithm for LCS is based on the anchoring technique, which previously appeared in classical LCS algorithms [88, 38, 13, 14, 23, 39, 40], as well as the quantum algorithm for $(1-\varepsilon)$-approximate LCS by Le Gall and Seddighin [51]. In this technique, one constructs a set of anchors, and only focuses on finding anchored length-d common substrings, defined as follows.

Definition 3.1.1 (Anchored common substrings and anchor sets)**.** For $T = S_1\$S_2$ of length $|T| = n$ and subset $\mathsf{C} \subseteq [1\mathinner{..}n]$, a common substring $S_1[i_1\mathinner{..}i_1+d) = S_2[i_2\mathinner{..}i_2+d)$ is said to be *anchored* by C, if there exists a shift $h \in [0\mathinner{..}d)$ such that $i_1+h, |S_1|+1+i_2+h \in \mathsf{C}$.

Given $T = S_1\$S_2$ and threshold length d, we say $\mathsf{C} \subseteq [1\mathinner{..}n]$ is an *anchor set* if the following holds: if $\mathrm{LCS}(S_1, S_2) \geq d$, then S_1 and S_2 must have a length-d common substring anchored by C.

The anchor set C may depend on T and d. We say C is $\mathcal{T}(n,d)$*-time constructible*, if there is a quantum algorithm that outputs $C(j) \in [n]$ in $\mathcal{T}(n,d)$ time given any index $1 \leq j \leq m$, and $\mathsf{C} = \{C(1), C(2), \dots, C(m)\}$.[1]

The set $[1\mathinner{..}n]$ is a trivial anchor set, but there are constructions of much smaller size. For example, several previous algorithms [35, 26, 53, 40, 51] used *difference covers* [35, 75] to obtain an $\widetilde{O}(1)$-time constructible anchor set of size $m = O(n/\sqrt{d})$, which is deterministic and oblivious to the content of the input strings S_1, S_2. The following theorem (which will be proved in Section 3.4) achieves a smaller size by a probabilistic non-oblivious construction that takes longer time to compute.

Theorem 3.1.2 (Anchor set construction)**.** *There is a* $d^{1/2+o(1)} \cdot \operatorname{poly}\log(n)$*-time constructible anchor set* C *of size* $m = O(n/d^{1-o(1)})$*. This set* C *depends on the input strings* S_1, S_2 *and* $\operatorname{poly}\log n$ *random coins, and has at least* $2/3$ *success probability.*

Let $\mathsf{C} = \{C(1), \dots, C(m)\} \subseteq [n]$ be an anchor set of size m. For every anchor

[1]The elements $C(1), C(2), \dots, C(m)$ are allowed to contain duplicates, and are not necessarily sorted in any particular order.

$C(k)$ indexed by $k \in [m]$, we associate it with a pair of strings $(P(k), Q(k))$, where

$$P(k) := T[C(k) \mathinner{.\,.} C(k)+d),$$
$$Q(k) := \big(T(C(k)-d \mathinner{.\,.} C(k)-1]\big)^R$$

are substrings (or reversed substrings) of T obtained by extending from the anchor $C(k)$ to the right or reversely to the left. The length of $P(k)$ is at most d, and the length of $Q(k)$ is at most $d-1$.[2] We say the string pair $(P(k), Q(k))$ is *red* if $C(k) \in [1 \mathinner{.\,.} |S_1|]$, or *blue* if $C(k) \in [|S_1|+1 \mathinner{.\,.} n]$. We also say $k \in [m]$ is a *red index* or a *blue index*, depending on the color of the string pair $(P(k), Q(k))$. Then, from the definition of anchor sets, we immediately have the following simple observation.

Proposition 3.1.3 (Witness pair)**.** *We have $LCS(S_1, S_2) \geq d$ if and only if there exist a red string pair $(P(k), Q(k))$ and a blue string pair $(P(k'), Q(k'))$ where $k, k' \in [m]$, such that $\mathsf{lcp}(P(k), P(k')) + \mathsf{lcp}(Q(k), Q(k')) \geq d$. In such case, (k, k') is called a* witness pair*.*

Proof. Suppose $LCS(S_1, S_2) \geq d$. Then the property of the anchor set C implies the existence of a shift $h \in [0 \mathinner{.\,.} d)$ and a length-d common substring $S_1[i \mathinner{.\,.} i+d) = S_2[j \mathinner{.\,.} j+d)$ such that $i + h = C(k), |S_1| + 1 + j + h = C(k')$ for some $k, k' \in [m]$. Then, we must have $\mathsf{lcp}(P(k), P(k')) \geq d-h$ and $\mathsf{lcp}(Q(k), Q(k')) \geq h$, implying that (k, k') is a witness pair.

Conversely, the existence of a witness pair immediately implies a common substring of length at least d. $\square$

In Section 3.2 and Section 3.3, we describe a quantum walk algorithm for finding a witness pair, and prove the following theorem.

Theorem 3.1.4. *Given input strings S_1, S_2 and threshold d, suppose there is a $\mathcal{T}$-time constructible anchor set of size m. Then, one can decide whether* $\mathrm{LCS}(S_1, S_2) \geq d$*, in*

$$\widetilde{O}(m^{2/3} \cdot (\sqrt{d} + \mathcal{T}))$$

[2] We use the convention $T[x \mathinner{.\,.} y] := T[\max\{1, x\} \mathinner{.\,.} \min\{y, |T|\}]$, which has shorter length when x or y is out of bound.

quantum time.

Our main theorem immediately follows from Theorem 3.1.2 and Theorem 3.1.4.

Theorem 1.2.1 (LCS with threshold d, upper bound)**.** *Given $S_1, S_2 \in \Sigma^n$, there is a quantum algorithm that decides whether S_1, S_2 have a common substring of length d in $\widetilde{O}(n^{2/3}/d^{1/6-o(1)})$ quantum query complexity and time complexity.*

Proof. The statement follows from combining Theorem 3.1.4 and Theorem 3.1.2 with $\mathcal{T} = d^{1/2+o(1)} \cdot \operatorname{poly}\log(n)$ and $m = O(n/d^{1-o(1)})$. $\square$

3.2 Anchoring via Quantum Walks

We shall prove Theorem 3.1.4 by giving a quantum walk algorithm to search for a witness pair. In this section, we focus on the query complexity of this quantum walk algorithm, and defer the time-efficient implementation to the next section.

Definition of the Johnson graph. Recall that $\mathsf{C} = \{C(1), \ldots, C(m)\}$ is an anchor set of size $|C| = m$. We perform a quantum walk on the Johnson graph with vertex set $\binom{[m]}{r}$, where r is a parameter to be determined later. A vertex $K = \{k_1, k_2, \ldots, k_r\} \subseteq [m]$ in the Johnson graph is called a *marked vertex*, if and only if $\{k_1, k_2, \ldots, k_r\}$ contains a witness pair (Proposition 3.1.3). If S_1 and S_2 have a common substring of length d, then at least $\binom{m-2}{r-2}/\binom{m}{r} = \Omega(r^2/m^2)$ fraction of the vertices are marked. Otherwise, there are no marked vertices.

Associated data. In the quantum walk algorithm, each state $K = \{k_1, \ldots, k_r\} \subseteq [m]$ is associated with the following data.

- The indices $k_1, \ldots, k_r$ themselves.
- The corresponding anchors $C(k_1), \ldots, C(k_r) \in [n]$.
- An array $(k_1^P, \ldots, k_r^P)$, which is a permutation of $k_1, \ldots, k_r$, such that $P(k_i^P) \preceq P(k_{i+1}^P)$ for all $1 \leq i < r$.

- The LCP array $h_1^P, \ldots, h_{r-1}^P$, where $h_i^P = \mathsf{lcp}(P(k_i^P), P(k_{i+1}^P))$

- An array $(k_1^Q, \ldots, k_r^Q)$, which is a permutation of $k_1, \ldots, k_r$, such that $Q(k_i^Q) \preceq Q(k_{i+1}^Q)$ for all $1 \le i < r$.

- The LCP array $h_1^Q, \ldots, h_{r-1}^Q$, where $h_i^Q = \mathsf{lcp}(Q(k_i^Q), Q(k_{i+1}^Q))$.

Note that we stored the *lexicographical orderings* of the strings $P(k_1), \ldots, P(k_r)$ and $Q(k_1), \ldots, Q(k_r)$ (for identical substrings, we break ties by comparing the indices themselves), as well as the *LCP arrays* which include the length of the longest common prefix of every pair of lexicographically adjacent substrings. By Lemma 2.1.1, these arrays together uniquely determine the values of $\mathsf{lcp}\left(P(k_i), P(k_j)\right)$ and $\mathsf{lcp}\left(Q(k_i), Q(k_j)\right)$, for *every* pair of $i, j \in [r]$.[3]

In the checking step of the quantum walk algorithm, we decide whether the state is marked, by searching for a witness pair (Proposition 3.1.3) in $\{k_1, \ldots, k_r\}$. Note that the contents of the involved strings $\{P(k_i)\}_{i\in[r]}$, $\{Q(k_i)\}_{i\in[r]}$ are no longer needed in order to solve this task, as long as we already know their lexicographical orderings and the LCP arrays. This task is termed as the *Two String Families LCP* problem in the literature [38], formalized as below.

Two String Families LCP

Input: r red/blue pairs of strings $(P_1, Q_1), (P_2, Q_2), \ldots, (P_r, Q_r)$ of lengths $|P_i|, |Q_i| \le d$, which are represented by the lexicographical orderings of $P_1, \ldots, P_r$ and of $Q_1, \ldots, Q_r$, and their LCP arrays

Task: Decide if there exist a red pair (P, Q) and a blue pair (P', Q'), such that $\mathsf{lcp}(P, P') + \mathsf{lcp}(Q, Q') \ge d$.

We will show how to solve this task time-efficiently in Section 3.3. For now, we only consider the query complexity of the algorithm, and we have the following simple observation, due to the fact that our associated information already uniquely determines the LCP value of every pair.

[3]To better understand this fact, observe that they uniquely determine the *compact tries* of $P(k_1), \ldots, P(k_r)$ and of $Q(k_1), \ldots, Q(k_r)$, where the LCP of two strings equals the depth of the lowest common ancestor of the corresponding nodes in the compact trie.

Proposition 3.2.1 (Query complexity of checking step is zero)**.** *Using the associated data defined above, we can determine whether $\{k_1, \dots, k_r\} \subseteq [m]$ is a marked state, without making any additional queries to the input.*

Now, we consider the cost of maintaining the associated data when the subset $\{k_1, \dots, k_r\}$ undergoes insertion and deletion during the quantum walk algorithm.

Proposition 3.2.2 (Update cost)**.** *Assume the anchor set C is $\mathcal{T}$-time constructible. Then, each update step of the quantum walk algorithm has query complexity $\mathcal{U} = \widetilde{O}(\sqrt{d} + \mathcal{T})$.*

Proof. Let us consider how to update the associated data when a new index k is being inserted into the subset $\{k_1, \dots, k_r\}$. The deletion process is simply the reverse operation of insertion.

The insertion procedure can be summarized by the pseudocode in Algorithm 1. First, we compute and store $C(k)$ in time $\mathcal{T}$. Then we use a binary search to find the correct place to insert k into the lexicographical orderings $(k_1^P, \dots, k_r^P)$ (and $(k_1^Q, \dots, k_r^Q)$). Since the involved substrings have length d, each lexicographical comparison required by this binary search can be implemented in $\widetilde{O}(\sqrt{d})$ time by Observation 2.5.1. After inserting k into the list, we update the LCP array by computing its LCP values $h_{\mathsf{pre}}, h_{\mathsf{suc}}$ with two neighboring substrings, and removing (by "uncomputing") the LCP value h_{old} between their neighbors which were adjacent at first, in $\widetilde{O}(\sqrt{d})$ time (Observation 2.5.1). $\square$

Proposition 3.2.3 (Setup cost)**.** *The setup step of the quantum walk has query complexity $\mathcal{S} = \widetilde{O}(r \cdot (\sqrt{d} + \mathcal{T}))$.*

Proof. We can set up the initial state for the quantum walk by simply performing r insertions successively using Proposition 3.2.2. $\square$

Remark 3.2.4. Observe that, in the insertion procedure in Algorithm 1, Lines 2 and 4-7 can be implemented also in *time complexity* $\widetilde{O}(\sqrt{d} + \mathcal{T})$. The time-consuming steps in Algorithm 1 are those that actually modify the data. For example, in Lines

Algorithm 1: The insertion procedure

```
1  Given an index $k \in [m]$
2  Compute $C(k)$
3  Store the data $(k, C(k))$
4  Compute the rank $i$ of $P(k)$ among $P(k_1^P), \ldots, P(k_r^P)$
5  Compute $h_{\mathsf{pre}} = \mathsf{lcp}(P(k_{i-1}^P), P(k))$
6  Compute $h_{\mathsf{suc}} = \mathsf{lcp}(P(k_i^P), P(k))$
7  Compute $h_{\mathsf{old}} = \mathsf{lcp}(P(k_{i-1}^P), P(k_i^P))$
8  Update $(k_1^P, \ldots, k_r^P) \leftarrow (k_1^P, \ldots, k_{i-1}^P, k, k_i^P, \ldots, k_r^P)$
9  Update $(h_1^P, \ldots, h_{r-1}^P) \leftarrow (h_1^P, \ldots, h_{i-2}^P, h_{\mathsf{pre}}, h_{\mathsf{suc}}, h_i^P, \ldots, h_{r-1}^P)$
10 Update $(k_1^Q, \ldots, k_r^Q)$ and $(h_1^Q, \ldots, h_{r-1}^Q)$ similarly as in Lines 4-9
```

8 and 9, the insertion causes some elements in the array to shift to the right, and would take $O(r)$ time if implemented naively. Later in Section 3.3 we will describe appropriate data structures to implement these steps time-efficiently.

Finally, by Theorem 2.6.1, the query complexity of our quantum walk algorithm (omitting $\operatorname{poly}\log(n)$ factors) is

$$\begin{aligned} &\mathcal{S} + \sqrt{\frac{m^2}{r^2}} \cdot (\mathcal{C} + \sqrt{r} \cdot \mathcal{U}) \qquad (3.1) \\ &= r \cdot (\sqrt{d} + \mathcal{T}) + \frac{m}{r} \cdot \big(0 + \sqrt{r} \cdot (\sqrt{d} + \mathcal{T})\big) \\ &= m^{2/3} \cdot (\sqrt{d} + \mathcal{T}), \end{aligned}$$

by choosing $r = m^{2/3}$.

3.3 Time-efficient Implementation

3.3.1 Overview

In this section, we show the quantum walk algorithm from Section 3.2 can be implemented time-efficiently, and finish the proof of Theorem 3.1.4.

We have to measure the quantum walk costs $\mathcal{S}, \mathcal{C}, \mathcal{U}$ in terms of the *time complexity* instead of query complexity. We will achieve setup time $\mathcal{S} = \widetilde{O}(r(\sqrt{d} + \mathcal{T}))$, checking time $\mathcal{C} = \widetilde{O}(\sqrt{rd})$, and update time $\mathcal{U} = \widetilde{O}(\sqrt{d} + \mathcal{T})$. Plugging into (3.1), the overall time complexity of the quantum walk remains $\widetilde{O}(m^{2/3} \cdot (\sqrt{d} + \mathcal{T}))$.

As mentioned in Section 3.2, there are two parts in the described quantum walk algorithm that are time-assuming:

- Maintaining the arrays of associated data under insertions and deletions (Remark 3.2.4).
- Solving the Two String Families LCP problem in the checking step.

Now we give an overview of how we address these two problems.

Dynamic arrays under insertions and deletions. A natural solution to speed up the insertions and deletions is to maintain the arrays of using appropriate data structures, which support the required operations in $\widetilde{O}(1)$ time. This "quantum walk plus data structures" framework was first used in Ambainis' element distinctness algorithm [9], and has been applied to many time-efficient quantum walk algorithms (see the discussion in Section 1.4). However, as noticed by Ambainis [9, Section 6.2], such data structures have to satisfy the following requirements in order to be applicable in quantum walk algorithms.

1. The data structure needs to be *history-independent*, that is, the representation of the data structure in memory should only depend on the set of elements stored (and the random coins used) by the data structure, *not* on the sequence of operations leading to this set of elements.
2. The data structure should guarantee *worst-case* time complexity (with high probability over the random coins) per operation.

The first requirement guarantees that each vertex of the Johnson graph corresponds to a unique quantum state, which is necessary since having multiple possible states would destroy the interference during the quantum walk algorithm. This requirement rules out many types of self-balancing binary search trees[4] such as AVL Tree and Red-Black Tree.

[4]One exception is Treap.

The second requirement rules out data structures with amortized or expected running time, which may take very long time in some of the operations. The reason is that, during the quantum algorithm, each operation is actually applied to a superposition of many instances of the data structure, so we would like the time complexity of an operation to have a fixed upper bound that is independent of the particular instance being operated on.

Ambainis [9] designed a data structure satisfying both requirements based on hash tables and skip lists, which maintains a sorted list of items, and supports insertions, deletions, and searching in $\widetilde{O}(1)$ time with high probability. Buhrman, Loff, Patro, and Speelman [33] modified this data structure to also support indexing queries, which ask for the k^{th} item in the current list (see Lemma 3.3.2 below). Using this data structure to maintain the arrays in our quantum walk algorithm, we can implement the update steps and the setup steps time-efficiently.

Dynamic Two String Families LCP. The checking step of our quantum walk algorithm (Proposition 3.2.1) requires solving a *Two String Families LCP* instance with r string pairs of lengths bounded by d. We will not try to solve this problem from scratch for each instance, since it is not clear how to solve it significantly faster than the $\widetilde{O}(r)$-time classical algorithm [38, Lemma 3] even using quantum algorithms. Instead, we *dynamically maintain* the solution using some data structure, which efficiently handles each update step during the quantum walk where we insert one string pair (P, Q) into (and remove one from) the current Two String Families LCP instance. As mentioned in Section 1.3.1, the classical data structure for this task given by Charalampopoulos, Gawrychowski, and Pokorski [39] is not applicable here, since it violates both requirements mentioned above: it maintains a heavy-light decomposition of the compact tries of the input strings, and rebuilds them from time to time to ensure amortized poly $\log(n)$ time complexity. It is not clear how to implement this strategy in a history-independent way and with worst-case time complexity per operation.

Instead, we will design a different data structure that satisfies the history independence and worst-case update time requirements, and can solve the Two String Families LCP problem on the maintained instance in $\widetilde{O}(\sqrt{rd})$ quantum time. This time complexity is much worse than the poly $\log(n)$ time achieved by the classical data structure of [39], but is sufficient for our purpose. As mentioned in Section 1.3.1, one challenge is the lack of a *comparison-based* data structure for 2D range query that also satisfies the two requirements above. We remark that there exist comparison-based data structures with history-independence but only with expected time complexity (e.g., [27]). There also exist folklore data structures for *integer coordinates* that have history-independence and worst-case time complexity (e.g., Lemma 3.3.3). For the easier problem of 1-dimensional range query, there exist folklore data structures (e.g., Lemma 3.3.2) that satisfy all three requirements. To get around this issue, we will use a sampling procedure and a version of the Balls-and-Bins argument, which can effectively convert the involved non-integer coordinates into integer coordinates. Then, we are able to apply 2D range query data structures over integer coordinates. Details will be given in Section 3.3.3.

3.3.2 Basic Data Structures

In this section, we will review several existing constructions of classical history-independent data structures.

Let D be a classical data structure using $\widetilde{O}(1)$ many random coins r that maintains a dynamically changing data set S. We say D is *history-independent* if for each possible S and r, the data structure has a unique representation $D(S, \mathsf{r})$ in the memory. Furthermore, we say *D has worst-case update time $O(\mathcal{T})$ with high probability*, if for every possible S and update operation $S \to S'$, with high probability over r, the time complexity to update from $D(S, \mathsf{r})$ to $D(S', \mathsf{r})$ is $O(\mathcal{T})$. Similarly we can define worst-case query time with high probability.

Since our quantum walk algorithm is over the Johnson graph $\binom{[m]}{r}$, for consistency we will use r to denote the size of the data structure instances in the following statements.

Hash tables. We use hash tables to implement efficient lookup operations without using too much memory.

Lemma 3.3.1 (Hash tables)**.** *There is a history-independent data structure of size* $\widetilde{O}(r)$ *that maintains a set of at most* r *key-value pairs* $\{(\mathsf{key}_1, \mathsf{value}_1), (\mathsf{key}_2, \mathsf{value}_2), \dots, (\mathsf{key}_r, \mathsf{value}_r)\}$ *where* key_i*'s are distinct integers from* $[m]$*, and supports the following operations in worst-case* $\widetilde{O}(1)$ *time with high probability:*

- ***Lookup:*** *Given a* $\mathsf{key} \in [m]$*, find the* value *corresponding to* key *(or report that* key *is not present in the set).*

- ***Insertion:*** *Insert a key-value pair into the set.*

- ***Deletion:*** *Delete a key-value pair from the set.*

Proof. (Sketch) The construction is similar to [9, Section 6.2]. The hash table has r buckets, each with the capacity for storing $O(\log m)$ many key-value pairs. A pair $(\mathsf{key}, \mathsf{value})$ is stored in the $h(\mathsf{key})^{\text{th}}$ bucket, and the pairs inside each bucket are sorted in increasing order of keys. If some buckets overflow, we can collect all the leftover pairs into a separate buffer of size r and store them in sorted order. This ensures that any set of r key-value pairs has a unique representation in the memory. And, each basic operation can be implemented in $\operatorname{poly}\log(m)$ time, unless there is an overflow. Using an $O(\log m)$-wise independent hash function $h\colon [m] \to [r]$, for any possible r-subset of keys, with high probability none of the buckets overflow.[5] $\square$

Dynamic arrays. We will need a dynamic array that supports indexing, insertion, deletion, and some other operations.

The *skip list* [84] is a probabilistic data structure which is usually used as an alternative to balanced trees, and satisfies the history-independence property. Ambainis' quantum Element Distinctness algorithm [9] used the skip list to maintain a *sorted*

[5]We remark that Ambainis only used a fixed hash function $h(i) = \lfloor r \cdot i/m \rfloor$, which ensures the buckets do not overflow with high probability *over a random* r*-subset* $K \subseteq [m]$ *of keys.* Ambainis showed that this property is already sufficient for the correctness of the quantum walk algorithm. Here we choose to state a different version that achieves high success probability for *every fixed* r-subset of keys, merely for keeping consistency with later presentation.

array, supporting insertions, deletions, and binary search. In order to apply the skip list in the quantum walk, a crucial adaptation in Ambainis' construction is to show that the random choices made by the skip list can be simulated using $O(\log n)$-wise independent functions [9, Section 6.2], which only take $\operatorname{poly}\log(n)$ random coins to sample. In the recent quantum fine-grained reduction result by Buhrman, Loff, Patro, and Speelman [33, Section 3.2], they used a more powerful version of skip lists that supports *efficient indexing*. We will use this version of skip lists with some slight extension.

Lemma 3.3.2 (Dynamic arrays)**.** *There is a history-independent data structure of size $\widetilde{O}(r)$ that maintains an array of items $(\mathsf{key}_1, \mathsf{value}_1), (\mathsf{key}_2, \mathsf{value}_2), \ldots, (\mathsf{key}_r, \mathsf{value}_r)$ with distinct keys (note that neither the keys nor the values are necessarily sorted in increasing order), and supports the following operations with worst-case $\widetilde{O}(1)$ time complexity and high success probability:*

- ***Indexing:*** *Given an index $1 \le i \le r$, return the i^{th} item $(\mathsf{key}_i, \mathsf{value}_i)$*
- ***Insertion:*** *Given an index $1 \le i \le r+1$ and a new item, insert it into the array between the $(i-1)^{st}$ item and the i^{th} item (shifting later items to the right).*
- ***Deletion:*** *Given an index $1 \le i \le r$, delete the i^{th} item from the array (shifting later items to the left).*
- ***Location:*** *Given a key, return its position i in the array (i.e., $\mathsf{key}_i = \mathsf{key}$).*
- ***Range-minimum query:*** *Given $1 \le a \le b \le r$, return $\min_{a \le i \le b}\{\mathsf{value}_i\}$.*

Proof. (Sketch) We will use (a slightly modified version of) the data structure described in [33, Section 3.2], which extends the construction of [9, Section 6.2] to support *insertion, deletion, and indexing*. Their construction is a (bidirectional) skip list of the items, where a pointer (a "skip") from an item $(\mathsf{key}, \mathsf{value})$ to another item $(\mathsf{key}', \mathsf{value}')$ is stored in a hash table as a key-value pair $(\mathsf{key}, \mathsf{key}')$. To support efficient indexing, for each pointer they also store the *distance* of this skip, which is used during an indexing query to keep track of the current position after following the pointers (similar ideas were also used in, e.g., [83, Section 3.4]). After every insertion or deletion, the affected distance values are updated recursively, by decomposing a level-i skip into $O(\log n)$ many level-$(i-1)$ skips.

A difference between their setting and ours is that they always keep the array sorted in increasing order of value's, and the position of an inserted item is decided by its relative order among the values in the array, instead of by a given position $1 \le i \le r+1$. Nevertheless, it is straightforward to adapt their construction to our setting, by using the distance values of the skips to keep track of the current position, instead of by comparing the values of items.

Note that using the distance values we can also efficiently implement the *Location* operation in a reversed way compared to *Indexing*, by following the pointers backwards and moving up levels.

To implement the *range-minimum query* operations, we maintain the range minimum value of each skip in the skip list, in a similar way to maintaining the distance values of the skips. They can also be updated recursively after each update. Then, to answer a query, we can travel from the a^{th} item to the b^{th} by following the pointers (this is slightly trickier if $a \neq 1$, where we may first move up levels and then move down). $\square$

We also need a 2D range sum data structure for points with integer coordinates.

Lemma 3.3.3 (2D range sum)**.** *Let integer $N \le n^{O(1)}$. There is a history-independent data structure of size $\widetilde{O}(r)$ that maintains a multi-set of at most r points $\{(x_1, y_1), \ldots, (x_r, y_r)\}$ with integer coordinates $x_i \in [N], y_i \in [N]$, and supports the following operations with worst-case $\widetilde{O}(1)$ time complexity and high success probability:*

- ***Insertion:*** *Add a new point (x, y) into the multiset (duplicates are allowed).*

- ***Deletion:*** *Delete the point (x, y) from the multiset (if it appears more than once, only delete one copy of them).*

- ***Range sum:*** *Given $1 \le x_1 \le x_2 \le N, 1 \le y_1 \le y_2 \le N$, return the number of points (x, y) in the multiset that are in the rectangle $[x_1 \mathinner{..} x_2] \times [y_1 \mathinner{..} y_2]$.*

Proof. (Sketch) Without loss of generality, assume N is a power of two. We use a simple folklore construction that resembles a 2D segment tree (sometimes called 2D range tree or 2D radix tree). Define a class $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \cdots \cup \mathcal{C}_{\log N}$ of sub-segments of the segment $[1 \mathinner{..} N]$ as follows:

$$\begin{aligned}
\mathcal{C}_1 &= \{[1 \mathinner{..} N]\},\\
\mathcal{C}_2 &= \{[1 \mathinner{..} N/2], [N/2+1 \mathinner{..} N]\},\\
\mathcal{C}_3 &= \{[1 \mathinner{..} N/4], [N/4+1 \mathinner{..} 2N/4], [2N/4+1 \mathinner{..} 3N/4], [3N/4+1 \mathinner{..} N]\},\\
&\cdots\\
\mathcal{C}_{\log N} &= \{[1 \mathinner{..} 1], [2 \mathinner{..} 2], \ldots, [N \mathinner{..} N]\}.
\end{aligned}$$

Then it is not hard to see that every segment $[a \mathinner{..} b] \subseteq [1 \mathinner{..} N]$ can be represented as the disjoint union of at most $2 \log N$ segments in $\mathcal{C}$. Consequently, the query rectangle $[x_1 \mathinner{..} x_2] \times [y_1 \mathinner{..} y_2]$ can always be represented as the disjoint union of $O(\log^2 N)$ rectangles of the form $\mathcal{I} \times \mathcal{J}$ where $\mathcal{I}, \mathcal{J} \in \mathcal{C}$.

Hence, for every $\mathcal{I}, \mathcal{J} \in \mathcal{C}$ with *non-zero* range sum $s(\mathcal{I} \times \mathcal{J})$, we store this range sum into a hash table, indexed by the canonical encoding of $(\mathcal{I}, \mathcal{J})$. Then we can efficiently answer all the range-sum queries by decomposing the rectangles and summing up their stored range sums.

When a point (x, y) is updated, we only need to update the range sums of $\log^2 N$ many rectangles that are affected, since each $a \in [1 \mathinner{..} N]$ is only included by $\log N$ intervals in $\mathcal{C}$. We may also need to insert a new rectangle into the hash table, or remove a rectangle once its range sum becomes zero. □

Data structures in quantum walk. Ambainis [9] showed that a history independent classical data structure D with worst-case time complexity $\mathcal{T}$ (with high probability over the random coins r) can be applied to the quantum walk framework by creating a uniform superposition over all possible r, i.e., the data structure storing data S corresponds to the quantum state $\sum_{\mathsf{r}} |D(S, \mathsf{r})\rangle|\mathsf{r}\rangle$. During the quantum walk algorithm, each data structure operation is aborted after running for $\mathcal{T}$ time steps. By doing this, some components in the quantum state may correspond to malfunctioning data structures, but Ambainis showed that this will not significantly affect the behavior of the quantum walk algorithm. We do not repeat the error analysis here, but instead refer interested readers to the proof of [9, Lemma 5 and 6] (see also [33, Lemma 1 and 2]).

3.3.3 Applying the Data Structures

Now we will use the data structures described in Section 3.3.2 to implement our quantum walk algorithm from Section 3.2 time-efficiently.

Recall that C is the $\mathcal{T}$-quantum-time-constructible anchor set of size m (Definition 3.1.1). The states of our quantum walk algorithms are r-subsets $K = \{k_1, k_2, \ldots, k_r\} \subseteq [m]$, where each index $k \in K$ is associated with an anchor $C(k) \in [n]$, which specifies the color (red or blue) of k and the pair $(P(k), Q(k))$ of strings of lengths at most d. We need to maintain the lexicographical orderings $(k_1^P, \ldots, k_r^P)$ and LCP arrays $(h_1^P, \ldots, h_{r-1}^P)$, so that $P(k_1^P) \preceq P(k_2^P) \preceq \cdots \preceq P(k_r^P)$ and $h_i^P = \mathsf{lcp}(P(k_i^P), P(k_{i+1}^P))$, and similarly maintain $(k_1^Q, \ldots, k_r^Q)$, $(h_1^Q, \ldots, h_{r-1}^Q)$ for the strings $\{Q(k)\}_{k \in K}$.

For $k \in K$, we use $\mathsf{pos}^P(k)$ to denote the position i such that $k_i^P = k$, i.e., the lexicographical rank of $P(k)$ among all $P(k_1), \ldots, P(k_r)$. Similarly, let $\mathsf{pos}^Q(k)$ denote the position i such that $k_i^Q = k$.

We can immediately see that all the steps in the *update step* (Algorithm 1) of our quantum walk can be implemented time-efficiently. In particular, we use a hash table (Lemma 3.3.1) to store the anchor $C(k)$ corresponding to each $k \in K$, and use Lemma 3.3.2 to maintain the lexicographical orderings and LCP arrays under insertions and deletions. Each update operation on these data structures takes $\widetilde{O}(1)$ time. Additionally, these data structures allow us to efficiently compute some useful information, as summarized below.

Proposition 3.3.4. *Given indices $k, k' \in K$, the following information can be computed in $\widetilde{O}(1)$ time.*

1. *The anchor $C(k)$, the color of k, and the lengths $|P(k)|, |Q(k)| \le d$.*
2. *$\mathsf{pos}^P(k)$ and $\mathsf{pos}^Q(k)$.*
3. *$\mathsf{lcp}(P(k), P(k'))$ and $\mathsf{lcp}(Q(k), Q(k'))$.*

Proof. For Item 1, rather than use $\mathcal{T}$ time to compute $C(k)$ (Definition 3.1.1), we instead look up the value of $C(k)$ from the hash table. Then, $C(k) \in [n]$ determines the color of k and the string lengths.

For Item 2, we use the location operation of the dynamic array data structure (Lemma 3.3.2).

For Item 3, we first compute $i = \mathsf{pos}^P(k), i' = \mathsf{pos}^P(k')$, and assume $i < i'$ without loss of generality. Then, by Lemma 2.1.1, we can compute $\mathsf{lcp}(P(k), P(k')) = \mathsf{lcp}(P(k_i^P), P(k_{i'}^P)) = \min\{h_i^P, h_{i+1}^P, \dots, h_{i'-1}^P\}$ using a range-minimum query from Lemma 3.3.2. □

The remaining task is to efficiently implement the checking step, where we need to solve the Two String Families LCP problem in $\widetilde{O}(\sqrt{rd})$ time. The goal is to find a red index $k^{\mathsf{red}} \in K$ and a blue index $k^{\mathsf{blue}} \in K$, such that $\mathsf{lcp}(P(k^{\mathsf{red}}), P(k^{\mathsf{blue}})) + \mathsf{lcp}(Q(k^{\mathsf{red}}), Q(k^{\mathsf{blue}})) \ge d$. Now we give an outline of the algorithm for solving this task.

Algorithm 2: Solving the Two String Families LCP problem in the checking step

```
1 Grover-Search over red indices $k^{\mathsf{red}} \in K$, and integers $d' \in [0\,.\,.\,d]$
2     Find $\ell^P, r^P$ such that $\mathsf{lcp}(P(k_i^P), P(k^{\mathsf{red}})) \ge d'$ if and only if $\ell^P \le i \le r^P$.
3     Find $\ell^Q, r^Q$ such that $\mathsf{lcp}(Q(k_i^Q), Q(k^{\mathsf{red}})) \ge d - d'$ if and only if
        $\ell^Q \le i \le r^Q$.
4     if exists a blue index $k^{\mathsf{blue}} \in K$ such that
        $\mathsf{pos}^P(k^{\mathsf{blue}}) \in [\ell^P\,.\,.\,r^P], \mathsf{pos}^Q(k^{\mathsf{blue}}) \in [\ell^Q\,.\,.\,r^Q]$ then return True
5 return False
```

In Algorithm 2, we use Grover search to find a red index $k^{\mathsf{red}} \in K$ and an integer $d' \in [0\,.\,.\,d]$, such that there exists a blue index $k^{\mathsf{blue}} \in K$ with $\mathsf{lcp}(P(k^{\mathsf{red}}), P(k^{\mathsf{blue}})) \ge$

d' and $\mathsf{lcp}(Q(k^{\mathsf{red}}), Q(k^{\mathsf{blue}})) \geq d-d'$. The number of Grover iterations is $\widetilde{O}(\sqrt{|K| \cdot d}) = \widetilde{O}(\sqrt{rd})$, and we will implement each iteration in $\operatorname{poly}\log(n)$ time. By Lemma 2.1.1, all the strings $P(k)$ that satisfy $\mathsf{lcp}(P(k), P(k^{\mathsf{red}})) \geq d'$ form a *contiguous segment* in the lexicographical ordering $P(k_1^P) \preceq \cdots \preceq P(k_r^P)$. In Line 2, we find the left and right boundaries ℓ^P, r^P of this segment, using a binary search with Proposition 3.3.4 (Item 3). Line 3 is similar to Line 2. Then, Line 4 checks the existence of such a blue string pair. It is clear that this procedure correctly solves the Two String Families LCP problem. The only remaining problem is how to implement Line 4 efficiently.

Note that Line 4 can be viewed as a 2D orthogonal range query, where each 2D point is a blue string pair $(P(k), Q(k))$, with coordinates being strings to be compared in lexicographical order. We cannot simply replace the coordinates by their ranks $\mathsf{pos}^P(k)$ and $\mathsf{pos}^Q(k)$ among the r substrings in the current state, since their ranks will change over time. It is also unrealistic to replace the coordinates by their ranks among all the possible substrings $\{P(k)\}_{k\in[m]}$, since m could be much larger than the desired overall time complexity $n^{2/3}$. These issues seem to require our 2D range query data structure to be comparison-based, which is also difficult to achieve as mentioned before.

Instead, we will use a sampling technique, which effectively converts the non-integer coordinates into integer coordinates. At the very beginning of the algorithm (before running the quantum walk), we uniformly sample r distinct indices $x_1, x_2 \ldots, x_r \in [m]$, and sort them so that $P(x_1) \preceq P(x_2) \preceq \cdots \preceq P(x_r)$ (breaking ties by the indices), in $\widetilde{O}(r(\sqrt{d} + \mathcal{T}))$ total time (this complexity is absorbed by the time complexity of the setup step $\mathcal{S} = O(r(\sqrt{d} + \mathcal{T}))$). Then, during the quantum walk algorithm, when we insert an index $k \in [m]$ into K, we assign it an *integer label* $\rho^P(k)$ defined as the unique $i \in [0\,..\,r]$ satisfying $P(x_i) \preceq s' \prec P(x_{i+1})$, which can be computed in $\widetilde{O}(\sqrt{d})$ time by a binary search on the sorted sequence $P(x_1) \preceq \cdots \preceq P(x_r)$. We also sample $y_1, \ldots, y_r \in [m]$ and sort them so that $Q(y_1) \preceq Q(y_2) \preceq \cdots \preceq Q(y_r)$, and similarly define the integer labels $\rho^Q(k)$. Intuitively, the (scaled) label $\rho^P(k) \cdot (m/r)$ estimates the rank of $P(k)$ among all the strings $\{P(k')\}_{k'\in[m]}$.

The following lemma formalizes this intuition. It states that in a typical r-subset $K = \{k_1, k_2, \dots, k_r\} \subseteq [m]$, not too many indices can receive the same label.

Lemma 3.3.5. *For any $c > 1$, there is a $c' > 1$, such that the following statement holds:*

For positive integers $r \le m$, let $A, B \subseteq [m]$ be two independently uniformly random r-subsets. Let $A = \{a_1, a_2, \dots, a_r\}$ where $a_1 < a_2 < \dots < a_r$, and denote

$$A_0 := [1\,..\,a_1), A_1 := [a_1\,..\,a_2), \dots, A_{r-1} := [a_{r-1}\,..\,a_r), A_r := [a_r\,..\,m].$$

Then,

$$\Pr_{A,B}\left[|A_i \cap B| \ge c' \log m \ \ \text{for some } 0 \le i \le r\right] \le \frac{1}{m^c}.$$

Proof. Let $k = c' \log m$ for some $c' > 1$ to be determined later, and we can assume $k \le r$. Observe that, $|A_i \cap B| \ge k$ holds for some i only if there exist $b, b' \in [m]$, such that $|[b\,..\,b'] \cap B| \ge k$ and $[b+1\,..\,b'] \cap A = \emptyset$.

Let $b, b' \in [m], b \le b'$. For $b' - b \ge (c+2)(m \ln m)/r$, we have

$$\Pr_A[[b+1\,..\,b'] \cap A = \emptyset] = \frac{\binom{m-(b'-b)}{r}}{\binom{m}{r}} \le \left(1 - \frac{b'-b}{m}\right)^r \le 1/m^{c+2}.$$

For $b' - b < (c+2)(m \ln m)/r$, we have

$$\begin{aligned}
\Pr_B\left[|[b\,..\,b'] \cap B| \ge k\right] &\le \frac{\binom{b'-b+1}{k} \cdot \binom{m-k}{r-k}}{\binom{m}{r}} \\
&= \binom{b'-b+1}{k} \cdot \frac{\binom{r}{k}}{\binom{m}{k}} \\
&\le \left(\frac{e(b'-b+1)}{k}\right)^k \cdot \left(\frac{r}{m}\right)^k \\
&< \left(\frac{e(c+3)\ln m}{k}\right)^k \\
&< 1/m^{c+3},
\end{aligned}$$

where we set $k = c' \log m = 3(c+3) \log m$.

The proof then follows from a union bound over all pairs of $b, b' \in [m]$. □

Then, we can use the 2D point $(\rho^P(k), \rho^Q(k))$ with integer coordinates to represent the string pair $(P(k), Q(k))$, and use the data structure from Lemma 3.3.3 to handle the 2D range sum queries. To correctly handle the points near the boundary of a query, we need to check them one by one, and Lemma 3.3.5 implies that in average case this brute force step is not expensive.

The pseudocode in Algorithm 3 describes the additional steps to be performed during each insertion step of the quantum walk (the deletion step is simply the reversed operation of the insertion step).

Algorithm 3: Extra steps in the insertion procedure (in addition to the steps in Algorithm 1)

```
1 Given an index $k \in [m]$
2 Compute the integer labels $\rho^P(k)$ and $\rho^Q(k)$ using binary search, and store
    them in hash table
3 if $k$ is blue then
4     Insert the 2D point $(\rho^P(k), \rho^Q(k))$ into the 2D range sum data structure
```

The pseudocode in Algorithm 4 describes how to implement Line 4 in Algorithm 2 for solving the Two String Families LCP problem. Line 4 correctly handles all the "internal" blue pairs $(P(k^{\mathsf{blue}}), Q(k^{\mathsf{blue}}))$, which must satisfy $\mathsf{pos}^P(k^{\mathsf{blue}}) \in [\ell^P, r^P]$ and $\mathsf{pos}^Q(k^{\mathsf{blue}}) \in [\ell^Q, r^Q]$ by the definition of our integer labels $\rho^P(\cdot), \rho^Q(\cdot)$ and Lines 2 and 3. In Line 5 we handle the remaining possible blue pairs, which must have $\rho^P(k^{\mathsf{blue}}) \in \{\widetilde{\ell}^P, \widetilde{r}^P\}$ or $\rho^Q(k^{\mathsf{blue}}) \in \{\widetilde{\ell}^Q, \widetilde{r}^Q\}$, and can be found by binary searches on the lexicographical orderings (to be able to do this, we need to maintain the lexicographical orderings of $P(k_1), \dots, P(k_r)$ and the sampled strings $P(x_1), \dots, P(x_r)$ *combined*).

Note that in Line 7 of Algorithm 4 we abort if we have checked more than $4c' \log m$ boundary points, so that Algorithm 4 has worst-case $\widetilde{O}(1)$ overall running time. But this early stopping would also introduce (one-sided) error if there are too many boundary points which we have no time to check. However, a straightforward application of Lemma 3.3.5 implies that, with high success probability over the initial samples $P(x_1) \preceq P(x_2) \preceq \cdots \preceq P(x_r)$ and $Q(y_1) \preceq Q(y_2) \preceq \cdots \preceq Q(y_r)$, only $1/\operatorname{poly}(m)$

Algorithm 4: Implementation of Line 4 in Algorithm 2

```
1 Given indices $\ell^P, r^P, \ell^Q, r^Q$.
2 Let $\widetilde{\ell}^P := \rho^P(k^P_{\ell^P}), \widetilde{r}^P := \rho^P(k^P_{r^P})$.
3 Let $\widetilde{\ell}^Q := \rho^Q(k^Q_{\ell^Q}), \widetilde{r}^Q := \rho^Q(k^Q_{r^Q})$.
4 if the 2D range sum of $[\widetilde{\ell}^P + 1 \mathinner{.\,.} \widetilde{r}^P - 1] \times [\widetilde{\ell}^Q + 1 \mathinner{.\,.} \widetilde{r}^Q - 1]$ is non-zero then
    return True
5 for blue index $k^{\mathsf{blue}} \in K$ such that $\rho^P(k^{\mathsf{blue}}) \in \{\widetilde{\ell}^P, \widetilde{r}^P\}$ or $\rho^Q(k^{\mathsf{blue}}) \in \{\widetilde{\ell}^Q, \widetilde{r}^Q\}$
    do
6       if $\mathsf{pos}^P(k^{\mathsf{blue}}) \in [\ell^P \mathinner{.\,.} r^P], \mathsf{pos}^Q(k^{\mathsf{blue}}) \in [\ell^Q \mathinner{.\,.} r^Q]$ then return True
7       if already looped $4c' \log m$ times then exit for loop
8 return False
```

fraction of the r-subsets $K = \{k_1, \dots, k_r\} \in [m]$ in the Johnson graph can have more than $c' \log m$ strings receiving the same label. On these problematic states $K = \{k_1, \dots, k_r\} \in [m]$, the checking procedure may erroneously recognize K as unmarked, while other states are handled correctly by Algorithm 4 since there is no early aborting. This decreases the fraction of marked states in the Johnson graph by only a $1/\operatorname{poly}(m)$ fraction, which does not affect the overall time complexity of our quantum walk algorithm.

3.4 Anchor Sets from String Synchronizing Sets

In this section, we describe the anchor sets claimed in Theorem 3.1.2. The construction crucially relies on the quantum algorithm for string synchronizing sets (Definition 1.2.4) from Theorem 1.2.5, which will be proved in Chapter 5. It also uses the Lexicographically Minimal String Rotation algorithm from Theorem 1.2.3 (which will be proved in Chapter 4) for computing Lyndon Roots. The arguments to construct anchor sets from string synchronizing sets mostly follow from the classical algorithm by Charalampopoulos, Kociumaka, Pissis, and Radoszewski [40].

For threshold length d, we set parameter $\tau = \lfloor d/3 \rfloor$. Define a *$\tau$-run* to be a run of length at least $3\tau - 1$ with period at most $\frac{1}{3}\tau$. Consider the following set B of positions in $T = S_1 \$ S_2$, where $|T| = n$.

Definition 3.4.1 ([40])**.** Define $\mathsf{B} \subseteq [n]$ as follows: for all τ-runs $T[l \mathinner{.\,.} r]$, let P be the Lyndon root of $T[l \mathinner{.\,.} r]$, and $P = T[i^{(b)} \mathinner{.\,.} i^{(b)} + p) = T[i^{(e)} \mathinner{.\,.} i^{(e)} + p)$ be the first and last occurrences of P in $T[l \mathinner{.\,.} r]$. Add $i^{(b)}, i^{(b)} + p$, and $i^{(e)}$ into B.

Then, [40] showed the following lemma.

Lemma 3.4.2 ([40, Lemma 15])**.** *Let A be a τ-string synchronizing set of $T = S_1\$S_2$. Then, $\mathsf{A} \cup \mathsf{B}$ is an anchor set.*

More specifically, let $S_1[i_1 \mathinner{.\,.} i_1{+}d) = S_2[i_2 \mathinner{.\,.} i_2{+}d)$ be a length-d common substring with minimized $i_1 + i_2$. Then, $S_1[i_1 \mathinner{.\,.} i_1 + d) = S_2[i_2 \mathinner{.\,.} i_2 + d)$ is anchored by $\mathsf{A} \cup \mathsf{B}$.

Now we are ready to describe the construction of our anchor set.

Definition 3.4.3 (Anchor set C)**.** Let A be the τ-synchronizing set of T determined by random seed σ (Theorem 1.2.5). Let $f(\tau) = \tau^{o(1)}$ be the expected sparsity upper bound in Theorem 1.2.5.

Define $\mathsf{J} = \{1, 1 + \tau, 1 + 2\tau, \dots\} \cap [n - 3\tau + 2]$ of size $|\mathsf{J}| \le O(n/\tau)$. For every $i \in \mathsf{J}$, let $\mathsf{C}_i \subseteq [1 \mathinner{.\,.} n]$ be defined by the following procedure.

- **Step 1:** If $|\mathsf{A} \cap [i \mathinner{.\,.} i{+}\tau)| \le 100 f(\tau)$, then add all the elements from $A \cap [i \mathinner{.\,.} i{+}\tau)$ into C_i. Otherwise, do not add any.

- **Step 2:** If $i \in \mathsf{Q}$ (defined in (5.1)), then let $p := \mathsf{per}(T[i \mathinner{.\,.} i + \tau)) \le \tau/3$, and extend this period to both directions (up to distance τ):

$$r := \max\left\{r : r \le \min\{n, i + 2\tau\} \wedge \mathsf{per}(T[i \mathinner{.\,.} r]) = p\right\},$$
$$l := \min\left\{l : l \ge \min\{1, i - \tau\} \wedge \mathsf{per}(T[l \mathinner{.\,.} i + \tau)) = p\right\}.$$

  Let P be the Lyndon root of $T[l \mathinner{.\,.} r]$. Let $P = T[i^{(b)} \mathinner{.\,.} i^{(b)}{+}p) = T[i^{(e)} \mathinner{.\,.} i^{(e)}{+}p)$ be the first and last occurrences of P in $T[l \mathinner{.\,.} r]$. Add $i^{(b)}, i^{(b)} + p$, and $i^{(e)}$ into C_i.

Finally, the anchor set C is defined as $\bigcup_{i \in \mathsf{J}} \mathsf{C}_i$.

Now we show that the anchor set C from Definition 3.4.3 satisfies the properties required by Theorem 3.1.2.

Proof of Theorem 3.1.2. Recall $n = |T| = |S_1| + 1 + |S_2|$ and $\tau = \lfloor d/3 \rfloor$. We first observe that $|\mathsf{C}_i| \le \tau^{o(1)}$ and $\mathsf{C} = \bigcup_{i \in \mathsf{J}} \mathsf{C}_i$ has size $|\mathsf{C}| \le |\mathsf{J}| \cdot \tau^{o(1)} \le n/\tau^{1-o(1)}$ by definition. Then, observe that C_i can be computed in $\tau^{1/2+o(1)} \cdot \operatorname{poly}\log(n)$ quantum time given any $i \in \mathsf{J}$, due to the efficient computability of A (Theorem 1.2.5), and fast quantum algorithms for computing period (Corollary 2.5.3) and minimal string rotation (Theorem 1.2.3). Hence, C is $\tau^{1/2+o(1)} \cdot \operatorname{poly}\log(n)$-time constructible.

It remains to explain why C is an anchor set with at least constant probability.

First we show that $\mathsf{B} \subseteq \mathsf{C}$. Let any τ-run $T[l \mathinner{.\,.} r]$ be given, with $P = T[i^{(b)} \mathinner{.\,.} i^{(b)} + p) = T[i^{(e)} \mathinner{.\,.} i^{(e)} + p)$ being the first and last occurrences of its Lyndon root P in $T[l \mathinner{.\,.} r]$. By definition of Q, we have $[l \mathinner{.\,.} r - \tau + 1] \subseteq \mathsf{Q}$. Let j_1, j_2 be the minimum and maximum $j \in \mathsf{J} \cap [l \mathinner{.\,.} r - \tau + 1]$ (which must be non-empty). Then, we must have $j_1 - l \in [0 \mathinner{.\,.} \tau)$ which then implies $i^{(b)}, i^{(b)} + p \in \mathsf{C}_{j_1}$ in Step 2 of Definition 3.4.3. Similarly, we can show $i^{(e)} \in \mathsf{C}_{j_2}$. Hence, $\mathsf{B} \subseteq \mathsf{C}$.

Now, let $S_1[i_1 \mathinner{.\,.} i_1 + d) = S_2[i_2 \mathinner{.\,.} i_2 + d)$ be a length-d common substring with minimized $i_1 + i_2$. If $S_1[i_1 \mathinner{.\,.} i_1 + d) = S_2[i_2 \mathinner{.\,.} i_2 + d)$ is already anchored by B, then we are done. Otherwise, by Lemma 3.4.2 it is anchored by $\mathsf{A} \cap \mathsf{B}$, for any τ-synchronizing set A. We consider the synchronizing positions in A that may be used as the anchor here, i.e., they are close to i_1 or $|S_1| + 1 + i_2$ up to $O(\tau)$ distance. By the sparsity property (Theorem 1.2.5), the expectation (over seed σ) of

$$|[i_1 - \tau \mathinner{.\,.} i_1 + d + \tau) \cap \mathsf{A}| + |[|S_1| + 1 + i_2 - \tau \mathinner{.\,.} |S_1| + 1 + i_2 + d + \tau) \cap \mathsf{A}|$$

is at most $10 \cdot f(\tau)$. By Markov's inequality, with constant probability, this is at most $100 f(\tau)$, meaning that these anchors will all be included in C by the Step 1 of Definition 3.4.3. Hence, C is an anchor set with at least constant probability. $\square$

3.5 Quantum Query Lower Bound for LCS with Threshold d

In this section, we observe a nearly matching lower bound for the LCS problem with threshold d.

Theorem 1.2.2 (LCS with threshold d, lower bound)**.** *For $|\Sigma| \geq \Omega(n/d)$, deciding whether $S_1, S_2 \in \Sigma^n$ have a common substring of length d requires $\Omega(n^{2/3}/d^{1/6})$ quantum queries.*

Our proof is based on known results in quantum query complexity. Let

$$P_d := \{x_1 x_2 \cdots x_d \in (\Sigma \cup \{\star\})^d : \text{ exactly one } i \in [d] \text{ satisfies } x_i \neq \star\}.$$

Define a partial function $\mathsf{PSEARCH}_d \colon P_d \to \Sigma$ by letting $\mathsf{PSEARCH}_d(x_1 \cdots x_d)$ return the only non-$\star$ symbol x_i ($i \in [d]$). We will use the following composition theorem with inner function being the $\mathsf{PSEARCH}$ problem [30].

Theorem 3.5.1 ([30])**.** *Let $f \colon \Sigma^m \to A$ be a function with quantum query complexity $Q(f)$. Let h be the composition of f with $\mathsf{PSEARCH}_d$, namely*

$$\begin{aligned} &h(x_{1,1}, \ldots, x_{1,d};\ \ldots\ ; x_{m,1}, \ldots, x_{m,d}) \\ &:= f(\mathsf{PSEARCH}_d(x_{1,1}, \ldots, x_{1,d}), \ldots, \mathsf{PSEARCH}_d(x_{m,1}, \ldots, x_{m,d})). \end{aligned}$$

Then, the quantum query complexity of h satisfies $Q(h) \geq \Omega(Q(f) \cdot \sqrt{d})$.

We will reduce the composition of (bipartite) Element Distinctness with $\mathsf{PSEARCH}_d$ to the LCS problem with threshold d.

Proof of Theorem 1.2.2. In the (bipartite) Element Distinctness problem, we are given two length-m arrays $a, b \in \Sigma^m$, and want to decide whether there exist $i, j \in [m]$ such that $a_i = b_j$. When $|\Sigma| \geq \Omega(m)$, this problem requires $\Omega(m^{2/3})$ quantum query complexity [3, 8].

Let $m = n/d$. Let $a, b \in (\Sigma \cup \{\star\})^{md}$ be an instance of Bipartite Element Distinctness problem composed with the $\mathsf{PSEARCH}_d$ problem. By Theorem 3.5.1, this problem requires quantum query complexity $\Omega(m^{2/3} \cdot \sqrt{d}) = \Omega(n^{2/3}/d^{1/6})$.

Now we will create an LCS instance of two strings S, T of length $\Theta(n)$. Define string S as the concatenation

$$S := A_1 \#_1 A_2 \#_2 \cdots \#_{m-1} A_m,$$

where $\#_i$ are distinct delimiter symbols, and block A_i ($i \in [m]$) is defined as

$$A_i = \star^{10d} a_{i,1} a_{i,2} \ldots a_{i,d} \star^{10d} .$$

Notice that, when $a_{i,1} a_{i,2} \ldots a_{i,d}$ is an input for the $\mathsf{PSEARCH}_d$ problem, the unique non-$\star$ character inside block A_i has at least $10d$ $\star$s next to it on both sides. The string T is similarly defined based on b, using a different set of delimter symbols.

If there exits $i, j \in [m]$ such that $\widetilde{a}_i = \widetilde{b}_j \neq \star$, where $\widetilde{a}_i = \mathsf{PSEARCH}_d(a_{i,1}, \ldots, a_{i,d})$, and $\widetilde{b}_j = \mathsf{PSEARCH}_d(b_{j,1}, \ldots, b_{j,d})$, then the LCS between S, T is at least $1 + 20d$, consisting of the matched non-$\star$ symbol together with $\star^{10d}$ on its left and another $\star^{10d}$ on its right. On the other hand, any substring in S (or T) of length at least $11d$ that contains no delimters must contain a non-$\star$ symbol, so a common substring between S and T of length at least $11d$ implies that a and b have a common non-$\star$ symbol.

Hence, we can solve the instance by checking whether $\mathrm{LCS}(S, T) \geq 20d + 1$ or $\mathrm{LCS}(S, T) \leq 11d - 1$. $\square$

Observe that the proof easily extends to $(2 - \varepsilon)$-approximation for any constant $\varepsilon > 0$. By a suitable binary encoding of Σ, one can extend the lower bound to the case of binary strings with a $O(\log n)$-factor loss, similar to [51].

Chapter 4

Minimal String Rotation

In this chapter, we prove Theorem 1.2.3 by designing efficient quantum algorithms for (Lexicographically) Minimal String Rotation, Minimal Suffix, and Maximal Suffix.

4.1 Minimal Length-ℓ Substrings

Rather than work with the Minimal String Rotation problem directly, we present an algorithm for the following problem, which is more amenable to work with using our divide-and-conquer approach.

Minimal Length-ℓ Substrings

Input: A string $S[1\mathinner{.\,.}n]$ and an integer $n/2 \le \ell \le n$

Task: Output all elements in $\arg\min_{1\le i\le n-\ell+1} S[i\mathinner{.\,.}i+l)$ represented as an arithmetic progression.

The elements in the output are guaranteed to be an arithmetic progression thanks to Lemma 2.1.3.

We will prove the following theorem.

Theorem 4.1.1. *Minimal Length-ℓ Substrings can be solved by a quantum algorithm with $n^{1/2+o(1)}$ query complexity and time complexity.*

For convenience, we also introduce the following problem.

Maximal String Rotation

Input: A string S

Task: Output a position $i \in [1\,..\,|S|]$ such that $S[j\,..\,|S|]S[1\,..\,j-1] \preceq S[i\,..\,|S|]s[1\,..\,i-1]$ holds for all $j \in [1\,..\,|S|]$. If there are multiple solutions, output the smallest such i.

We now use a series of simple folklore reductions to show that the Minimal Length-ℓ Substrings problem generalizes the Minimal String Rotation problem.

Proposition 4.1.2. *1. The Minimal String Rotation problem reduces to the Maximal String Rotation problem.*

2. The Maximal String Rotation problem reduces to the Maximal Suffix problem.

3. The Maximal Suffix problem reduces to the Minimal Suffix problem.

4. The Minimal Suffix problem reduces to the Minimal Length-ℓ Substrings problem.

Proof. **Item 1.** Given $S \in \Sigma^n$, where Σ is identified with the integer set $[1\,..\,|\Sigma|]$, let $T = \varphi(S[1]) \cdots \varphi(S[n])$, where $\varphi(c) := |\Sigma| - c + 1$. Observe that the lexicographically maximal rotation of T corresponds to the lexicographically minimal rotation of S.

Item 2. Given an input string $S \in \Sigma^n$ of the Maximal String Rotation problem, let $T = SS$ be the string of length $2n$ formed by concatenating S with itself. Suppose $i \in [1\,..\,n]$ is the smallest starting index of the maximal rotation of S. Then observe that i is the starting index of the maximal suffix of T as well.

Item 3. Given an input string $S \in \Sigma^n$ of the Maximal Suffix problem, let $\$ = |\Sigma| + 1$ denote a character lexicographically after all the characters in Σ, and define $T = \varphi(S[1])\varphi(S[2]) \cdots \varphi(S[n])\$$, where $\varphi(c) := |\Sigma| - c + 1$. Suppose that $S[i\,..\,n]$ is the maximal suffix of S. Then observe that $T[i\,..\,n+1]$ is the minimal suffix of T.

Item 4. Given an input string $S \in \Sigma^n$ of the Minimal Suffix problem, define string $T = S\,\underbrace{00\cdots0}_{n-1 \text{ times}}$ of length $2n-1$, where character 0 is smaller than every character from the alphabet Σ of S. Suppose i is the starting index of the minimal suffix of S. Then, i is also the unique index returned by solving the Minimal Length-n Substrings problem on T (note that n is at least half the length of T), due to the following simple fact: for $i, j \in [1\mathinner{.\,.} n], i \neq j$, we have $S[i\mathinner{.\,.} n] \prec S[j\mathinner{.\,.} n]$ if and only if $S[i\mathinner{.\,.} n]\underbrace{00\cdots0}_{i-1 \text{ times}} \prec S[j\mathinner{.\,.} n]\underbrace{00\cdots0}_{j-1 \text{ times}}$. □

By chaining the above reductions together, we prove our main theorem as a corollary of Theorem 4.1.1.

Proof of Theorem 1.2.3. By combining the reductions in Proposition 4.1.2, we see that all the problems mentioned in the theorem statement reduce to the Minimal Length-ℓ Substrings problem. These reductions are *local*: for an input string S and its image T under any of these reductions, any query to a character of T can be simulated with $O(1)$ queries to the characters of S. Thus, we can get a $n^{1/2+o(1)}$ query and time quantum algorithm for each of the listed problems by using the algorithm of Theorem 4.1.1 and simulating the aforementioned reductions appropriately in the query model. □

Remark 4.1.3. We remark that, from the $\Omega(\sqrt{n})$ quantum query lower bound for Minimal String Rotation [92], this chain of reductions also implies that Maximal Suffix and Minimal Suffix require $\Omega(\sqrt{n})$ quantum query complexity.

It remains to prove Theorem 4.1.1. To solve the Minimal Length-ℓ Substrings problem, it suffices to find any individual solution

$$i \in \mathop{\arg\min}_{1 \le i \le n-\ell+1} S[i\mathinner{.\,.} i+\ell),$$

and then use the quantum Exact String Matching algorithm to find all the elements (represented as an arithmetic progression) in $\widetilde{O}(\sqrt{n})$ time. Our approach will invoke the following "exclusion rule," which simplifies the previous approach used in [92]. We remark that similar kinds of exclusion rules have been applied previously in parallel algorithms for Exact String Matching [91] and Minimal String Rotation [57] (under the name of "Ricochet Property" or "duel"), as well as the quantum algorithm by Wang and Ying [92, Lemma 5.1]. The advantage of our exclusion rule is that it naturally yields a *recursive* approach for solving the Minimal Length-ℓ Substrings problem.

Lemma 4.1.4 (Exclusion rule)**.** *In the Minimal Length-ℓ Substrings problem with input $S[1\mathinner{.\,.} n]$ with $n/2 \le \ell \le n$, let*

$$I := \operatorname*{arg\,min}_{1\le i\le n-\ell+1} S[i\mathinner{.\,.} i+\ell)$$

denote the set of answers forming an arithmetic progression. For integers $a \ge 1, k \ge 1$ such that $a+k \le n-\ell+1$, let J denote the set of answers in the Minimal Length-k Substrings problem on the input string $S[a\mathinner{.\,.} a+2k)$. Then if $\{\min J, \max J\} \cap I = \emptyset$, we must have $J \cap I = \emptyset$.

Proof. First observe that

$$a + 2k - 1 \le n - \ell + k \le 2(n-\ell) \le n,$$

so $S[a\mathinner{.\,.} a+2k)$ is a length-$2k$ substring of S. Since the statement is trivial for $|J| \le 2$, we assume J consists of $j_1 < j_2 < \cdots < j_m$ where $m \ge 3$. Let $p = j_2 - j_1$. Then $p = (j_m - j_1)/(m-1) \le k/2$. Then from

$$S[j_1\mathinner{.\,.} j_1+k) = S[j_2\mathinner{.\,.} j_2+k) = \cdots = S[j_m\mathinner{.\,.} j_m+k)$$

we know that p must be a period of $S[j_1\mathinner{.\,.} j_m+k)$.[1] We consider the first position r where this period p stops, that is, $r := \min\{j_m + k \le r \le n : S[r] \ne S[r-p]\}$. If such r does not exist, let $r = n+1$. With this setup, we now proceed to prove the contrapositive of the original claim.

Suppose $j_q \in I$ for some $1 \le q \le m$. We consider three cases.

- **Case 1:** $r \ge j_m + \ell$.

  In this case, we must have $S[j_1\mathinner{.\,.} j_1+\ell) = S[j_2\mathinner{.\,.} j_2+\ell) = \cdots = S[j_m\mathinner{.\,.} j_m+\ell)$. Then, $j_q \in I$ implies $j_1 \in I$ and $j_2 \in I$.

[1] In fact, p is the minimum period of this substring.

- **Case 2:** $r < j_m + \ell$, and $S[r] < S[r-p]$.

  For every $1 \leq t \leq m-1$, by the definition of r, we must have $S[j_{t+1} \mathinner{.\,.} r) = S[j_t \mathinner{.\,.} r-p)$. Then from $S[r] < S[r-p]$ we have $S[j_t \mathinner{.\,.} j_t+\ell) \succeq s[j_{t+1} \mathinner{.\,.} j_{t+1}+\ell)$. Hence, $j_q \in I$ implies $j_{q+1}, j_{q+1}, \ldots, j_m \in I$.

- **Case 3:** $r < j_m + \ell$, and $S[r] > S[r-p]$.

  By an argument similar to Case 2, we can show $S[j_t \mathinner{.\,.} j_t+\ell) \preceq S[j_{t+1} \mathinner{.\,.} j_{t+1}+\ell)$. Then, $j_q \in I$ implies $j_{q-1}, j_{q-2}, \ldots, j_1 \in I$.

Thus $\{j_1, j_m\} \cap I \neq \emptyset$ in all of the cases, which proves the desired result. □

4.2 Divide and Conquer Algorithm

To motivate our quantum algorithm, we first describe a classical algorithm for the Minimal $n/2$-length Substring problem which runs in $O(n \log n)$ time (note that other classical algorithms can solve the problem faster in $O(n)$ time). Our quantum algorithm will use the same setup, but obtain a speed-up via Grover search. For the purpose of this overview, we assume n is a power of 2. The classical algorithm works as follows:

Suppose we are given an input string S of length n and target substring size $\ell = n/2$. Set $m = \ell/2 = n/4$. Then the first half of the solution (i.e. the first m characters of a minimum length ℓ-substring) are contained entirely in either the block $S_1 = S[1 \mathinner{.\,.} n/2]$ or the block $S_2 = S[n/4 \mathinner{.\,.} 3n/4)$.

With that in mind, we recursively solve the problem on the strings S_1 and S_2 with target size m in both cases. Let u_1 and v_1 be the smallest and largest starting positions returned by the recursive call to S_1 respectively. Define u_2 and v_2 as the analogous positions returned by the recursive call to S_2. Then by Lemma 4.1.4, the true starting position of the minimal ℓ-length substring of S is in $\{u_1, u_2, v_1, v_2\}$.

We identify the ℓ-length substrings starting at each of these positions, and find their lexicographic minimum in $O(n)$ time via linear-time string comparison. This lets us find at least one occurrence of the minimum substring of length ℓ. Then, to find all occurrences of this minimum substring, we use a linear time string matching algorithm (such as the classic Knuth-Morris-Pratt algorithm [68]) to find the first two occurrences of the minimum length ℓ substring in S. The difference between the starting positions then lets us determine the common difference of the arithmetic sequence of positions encoding all starting positions of the minimum substring.

If we let $\mathcal{T}(n)$ denote the runtime of this algorithm, the recursion above yields a recurrence

$$\mathcal{T}(n) = 2\mathcal{T}(n/2) + O(n)$$

which solves to $\mathcal{T}(n) = O(n \log n)$.

4.3 Quantum Speedup

Next, we show how to improve the runtime of this divide-and-conquer approach in the quantum setting. The key change is to break the string into b blocks, and apply quantum minimum finding over these blocks which only takes $\widetilde{O}(\sqrt{b})$ recursive calls, instead of b recursive calls needed by the classical algorithm. We will set b large enough to get a quantum speedup.

Proof of Theorem 4.1.1. Let b be some parameter to be set later. For convenience assume that b divides both ℓ and n (this assumption does not affect the validity of our arguments, and is only used to let us avoid working with floor and ceiling functions). Set $m = \ell/b$.

For each nonnegative integer $k \le \lfloor n/m \rfloor - 2$ we define the substring

$$S_k = S(km \mathinner{.\,.} (k+2)m].$$

Also set $S_{\lfloor n/m \rfloor - 1} = S(n - 2m \mathinner{.\,.} n]$.

These S_k blocks each have length $2m$, and together cover every substring of length m in S. Let P be the minimum length-ℓ substring in S. By construction, the first $m = \ell/b$ characters of P is contained entirely in one of the S_k blocks.

For each block S_k, let P_k denote its minimum length-m substring and let u_k and v_k be the smallest and largest starting positions respectively of an occurrence of P_k in S_k. The lexicographically smallest prefix P_k will make up the first m characters of the minimum length-ℓ substring. Thus by Lemma 4.1.4, we know the minimum length-ℓ substring of S must start at position u_k or v_k for some index k.

We now use quantum minimum finding to find P. We search over the $\Theta(n/m) = \Theta(b)$ blocks above. To compare blocks S_i and S_j, we recursively solve the Minimal Length-m Substrings problem on S_i and S_j to find positions u_i, v_i and u_j, v_j. Then we look at the substrings of length ℓ starting at these four positions. By binary search and Grover search (Observation 2.5.1), in $\widetilde{O}(\sqrt{n})$ time we can determine which of these four substrings is lexicographically the smallest. If the smallest of these substrings came from S_i we say block S_i is smaller than block S_j, and vice versa.

After running the minimum finding algorithm, we will have found P. To return all occurrences of P, we can then use the quantum algorithm for Exact String Matching to find the two leftmost occurrences and the rightmost occurrence of P in S in $\widetilde{O}(\sqrt{n})$ time. Together they determine the positions of all copies of P in S as an arithmetic sequence, which we can return to solve the original problem.

It remains to check the runtime of the algorithm. Let $T(n)$ denote the runtime of the algorithm with error probability at most $1/n$. Recall that our algorithm solves Minimum Finding over $\Theta(b)$ blocks, where each comparison involves a recursive call on strings of size $2m = \Theta(n/b)$ and a constant number of string comparisons of length n (via Observation 2.5.1), and finally solves Exact String Matching for strings of size $\Theta(n)$. Hence we have the recurrence (assuming all logarithms are base 2)

$$\mathcal{T}(n) \leq \widetilde{O}(\sqrt{b}) \cdot \left(\mathcal{T}(n/b) + \widetilde{O}(\sqrt{n})\right) + \widetilde{O}(\sqrt{n}) = c(\log b)^e \sqrt{b}\left(\mathcal{T}(n/b) + \sqrt{n}\right)$$

for some constants $c, e > 0$, where the polylogarithmic factors are inherited from the subroutines we use and the possibility of repeating our steps $O(\log n)$ times to drive down the error probability. Now set

$$b = 2^{d(\log n)^{2/3}}$$

for some constant d. We claim that for sufficiently large d, we recover a runtime of $\mathcal{T}(n) = n^{1/2} \cdot 2^{d(\log n)^{2/3}}$.

We prove this by induction. The result holds when n is a small constant by taking d large enough. Now, suppose we want to prove the result for some arbitrary n, and that the claimed runtime bound holds on inputs of size less than n. Then using the recurrence above and the inductive hypothesis we have

$$\begin{aligned}\mathcal{T}(n) &\leq c(\log b)^e\sqrt{b}\left(T(n/b) + \sqrt{n}\right)\\ &\leq c(\log b)^e\sqrt{n}\left(2^{d(\log(n/b))^{2/3}} + \sqrt{b}\right)\\ &\leq 2c(\log b)^e\sqrt{n}\cdot 2^{d(\log(n/b))^{2/3}},\end{aligned}$$

where the last inequality follows from $d(\log(n/b))^{2/3} \geq d(\log(\sqrt{n}))^{2/3} > \frac{1}{2}d(\log n)^{2/3} = \log(\sqrt{b})$ for large enough n. Equivalently, this means that

$$\frac{\mathcal{T}(n)}{n^{1/2}2^{d(\log n)^{2/3}}} \leq 2c \cdot 2^{e(\log\log b) - d\left((\log n)^{2/3} - (\log n - \log b)^{2/3}\right)}. \tag{4.1}$$

Using the mean value theorem, we can bound

$$\begin{aligned}(\log n)^{2/3} - (\log n - \log b)^{2/3} &\geq (2/3)(\log b)(\log n)^{-1/3}\\ &= (2/3)d(\log n)^{1/3}\\ &\geq \omega(\log\log b),\end{aligned}$$

where the last inequality follows from $\log\log b = \log d + (2/3)\log\log n$. Thus, by taking d to be a large enough constant in terms of c and e, we can force the right hand side of (4.1) to be less than 1, which proves that

$$\mathcal{T}(n) \leq n^{1/2}2^{d(\log n)^{2/3}}.$$

This completes the induction, and proves that we can solve the Minimum Length-ℓ Substrings problem in the desired runtime as claimed. $\square$

Chapter 5

String Synchronizing Sets

In this chapter, we prove Theorem 1.2.5 by designing an efficient quantum algorithm for constructing a string synchronizing set [63] (Definition 1.2.4).

5.1 Review of Kempa and Kociumaka's Construction

We first review the underlying framework of Kempa and Kociumaka's construction [63], which will be used in our construction as well. Define set

$$\mathsf{Q} := \{i \in [1 \mathinner{.\,.} n-\tau+1] : \mathsf{per}(T[i \mathinner{.\,.} i+\tau)) \le \tau/3\}, \tag{5.1}$$

which contains the starting positions of highly-periodic length-τ substrings.

Let $\phi\colon \Sigma^\tau \to X$ be a mapping from length-τ strings to some totally ordered set X. For index $i \in [1 \mathinner{.\,.} n-\tau+1]$ in the input string $T[1 \mathinner{.\,.} n]$, we introduce the shorthand

$$\Phi(i) := \phi(T[i \mathinner{.\,.} i+\tau)). \tag{5.2}$$

Then, define the synchronizing set as

$$\mathsf{A} := \big\{i \in [1 \mathinner{.\,.} n-2\tau+1] : \min\{\Phi(j) : j \in [i \mathinner{.\,.} i+\tau] \setminus \mathsf{Q}\} \in \{\Phi(i), \Phi(i+\tau)\}\big\}. \tag{5.3}$$

Then, Kempa and Kociumaka proved the following lemma.

Lemma 5.1.1 ([63, Lemma 8.2])**.** *The set* A *defined in* (5.3) *is always a* τ*-synchronizing set of* T*.*

It is straightforward to check that A satisfies the consistency property in Definition 1.2.4. To provide an intuition, we briefly restate their proof for the density property in the simpler case where $\mathsf{Q} = \emptyset$. Given i, pick $k \in [i \mathinner{.\,.} i + 2\tau)$ with the minimum $\Phi(k)$. If $k < i + \tau$, then $\Phi(k) = \min\{\Phi(j) : j \in [k \mathinner{.\,.} k + \tau]\}$, and hence $k \in \mathsf{A}$. Otherwise, $\Phi(k) = \min\{\Phi(j) : j \in [k - \tau \mathinner{.\,.} k]\}$, and hence $k - \tau \in \mathsf{A}$. In either case, $\mathsf{A} \cap [i \mathinner{.\,.} i + \tau) \neq \emptyset$.

From now on, we assume $\tau \geq 100$. For $\tau < 100$, simply setting $\mathsf{A} := [1 \mathinner{.\,.} n - 2\tau + 1] \setminus \mathsf{Q}$ already satisfies all the requirements in Theorem 1.2.5.

The following property on the structure of set Q will be needed later.

Lemma 5.1.2. *Suppose interval* $[l \mathinner{.\,.} r] \subseteq [1 \mathinner{.\,.} n - \tau + 1]$ *has length* $r - l + 1 \leq \tau/3$*. Then,* $[l \mathinner{.\,.} r] \cap \mathsf{Q}$ *is either empty or an interval.*

Proof. We only need to consider the case where $|[l \mathinner{.\,.} r] \cap \mathsf{Q}| \geq 2$, and let $j < k$ be the minimum and maximum element in $[l \mathinner{.\,.} r] \cap \mathsf{Q}$, respectively. By definition of Q, we have $p = \mathsf{per}(T[j \mathinner{.\,.} j + \tau - 1]) \leq \tau/3$ and $q = \mathsf{per}(T[k \mathinner{.\,.} k + \tau - 1]) \leq \tau/3$. Applying weak periodicity lemma (Lemma 2.1.2) on the substring $T[k \mathinner{.\,.} j + \tau - 1]$ of length $j + \tau - k \geq 2\tau/3$, we know $\gcd(p, q)$ is a period of $T[k \mathinner{.\,.} j + \tau - 1]$. So $\gcd(p, q)$ is also a period of $T[j \mathinner{.\,.} k + \tau - 1]$. This implies $[j \mathinner{.\,.} k] \subseteq \mathsf{Q}$, and hence $[l \mathinner{.\,.} r] \cap \mathsf{Q}$ equals the contiguous interval $[j \mathinner{.\,.} k]$. □

In the following sections, our goal is to design a suitable (randomized) mapping ϕ that makes A sparse in expectation and allows an efficient quantum algorithm to report elements in A. Our main tool for constructing ϕ is Vishkin's Deterministic Sampling, which we shall review in the following section.

5.2 Review of Vishkin's Deterministic Sampling

Vishkin's Deterministic Sampling is a powerful technique originally designed for parallel pattern matching algorithms [91]. Ramesh and Vinay [85] adapted this technique into a quantum algorithm for Exact Pattern Matching (Theorem 2.5.2).

The following version of deterministic sampling is taken from Wang and Ying's presentation [92] of Ramesh and Vinay's quantum pattern matching algorithm. Compared to the original classical version by Vishkin, the main differences are: (1) The construction takes a random seed σ (but it is called deterministic sampling nonetheless), and (2) it explicitly deals with periodic patterns.

Lemma 5.2.1 (Quantum version of Vishkin's deterministic sampling, [85, 92]). *Let round parameter $w = O(\log n)$. Given $S \in \Sigma^n$, every seed $\sigma \in \{0,1\}^w$ determines a* deterministic sample $\mathsf{DS}_\sigma(S)$ *consisting of an* offset δ *and a sequence of w* checkpoints $i_1, i_2, \ldots, i_w$, *with the following properties:*

- $\mathsf{DS}_\sigma(S) = (\delta; i_1, i_2, \ldots, i_w)$ *can be computed in $\widetilde{O}(\sqrt{n})$ quantum time given S and σ.*
- *The offset satisfies $\delta \in [0 \mathinner{.\,.} \lfloor n/2 \rfloor]$.*
- *The checkpoints satisfy $i_j - \delta \in [1 \mathinner{.\,.} n]$ for all $j \in [w]$.*
- *Given $S \in \Sigma^n$, the following holds with at least $1 - n/2^w$ probability over uniform random $\sigma \in \{0,1\}^w$: for every offset $\delta' \in [0 \mathinner{.\,.} \lfloor n/2 \rfloor]$ with $\delta' - \delta$ not being a multiple of $\mathsf{per}(S)$, $\mathsf{DS}_\sigma(S)$ contains a checkpoint i_j such that $i_j - \delta' \in [1 \mathinner{.\,.} n]$ and $S[i_j - \delta'] \neq S[i_j - \delta]$.*

  If the property above is satisfied, we say σ generates a successful *deterministic sample $\mathsf{DS}_\sigma(S)$ for S.*

For completeness, here we include the proof of Lemma 5.2.1 given by [92].

Proof of Lemma 5.2.1 based on [92]. The algorithm is summarized in Algorithm 5. It terminates in no more than $w + 1 = O(\log n)$ rounds. To implement this algorithm,

Algorithm 5: Constructing deterministic samples given $S \in \Sigma^n$ and $\sigma \in \{0,1\}^w$

```
1  begin
2      Initialize all offsets in $[0\,..\,\lfloor n/2 \rfloor]$ as *alive*;
3      Initialize empty checkpoint list $C$, and $\mathsf{flag} \leftarrow$ **false**;
4      while *true* do
5          $p, q \leftarrow$ minimum and maximum offsets alive, respectively ;
6          Find the minimum $i \in [1+p\,..\,n+q]$ such that $S[i-p] \neq S[i-q]$ ;
7          if *such i exists* then
8              Append $i$ to list $C$, and return "failure" if $\mathsf{length}(C) > w$;
9              Let $c_i \leftarrow \begin{cases} S[i-p], & \text{if } \sigma[\mathsf{length}(C)] = 0, \\ S[i-q], & \text{if } \sigma[\mathsf{length}(C)] = 1; \end{cases}$
10             Kill all offsets $\delta'$ with $S[i-\delta'] \neq c_i$ ;
11         else if $p = q$ *or* $\mathsf{flag} =$ *true* then
12             return *offset* $\delta := p$ *with checkpoint list* $C$
13         else
14             $\mathsf{flag} \leftarrow$ **true**;
15             Kill offset $q$;
```

observe that: whether an offset is killed or not can be decided by checking against all checkpoints currently in list C (Line 10) (and additionally Line 15 if $\mathsf{flag} = \textbf{true}$), in $\operatorname{poly}\log(n)$ time. Hence, Line 5 can be implemented using quantum minimum finding, in $\widetilde{O}(\sqrt{n})$ time. Line 6 can be implemented using Grover search with binary search. Hence, the overall quantum time complexity is $\widetilde{O}(\sqrt{n})$.

Observe that, at Line 10, over the random coin $\sigma[\mathsf{length}(C)] \in \{0,1\}$, in expectation at least half of the offsets that are currently alive will be killed by the current checkpoint. Hence, after finding w checkpoints, the expected number of offsets that are still alive is at most $\lfloor n/2 \rfloor / 2^w$. When there is only one offset alive, Line 11 successfully terminates. So by Markov's inequality, it reports "failure" with no more than $n/2^w$ probability.

It remains to show that the final property is satisfied if the algorithm successfully terminates.

First, notice that whenever an offset δ' is killed by Line 10, it holds for the checkpoint i that $S[i-\delta'] \neq c_i$, while $S[i-\delta''] = c_i$ holds for all the offsets δ'' that are still alive, in particular for the offset δ that will be returned finally. Hence, $S[i-\delta'] \neq S[i-\delta]$ for this checkpoint i.

Next, observe that whenever an offset δ' is killed by Line 10, all other offsets δ' with $\mathsf{per}(S) \mid (\delta - \delta')$ are also killed at this point. So we only need to consider congruence class of offsets modulo $\mathsf{per}(S)$. Note that the **if** check at Line 7 fails if and only if $\mathsf{per}(S) \mid (q-p)$.

If Line 11 is reached with $\mathsf{flag} = \mathbf{false}$, then $p = q = \delta$, and all other offsets $\delta' \neq \delta$ must have been killed by Line 10, so the desired property is satisfied.

Now consider the case where flag is set **true** at some point. Denote the current values of p, q by p_1, q_1, which belong to the same congruence class. Then, consider the next time where the **if** check at Line 7 fails, at which point the values of p, q are denoted p_2, q_2, which belong to the same congruence class. We know $p_1 \le p_2 \le q_2 < q_1$. Hence, the congruence classes of offsets in the interval $[p_1 \mathinner{..} p_2)$ have been killed by Line 10, and the congruence classes of offsets in the interval $[q_2 \mathinner{..} q_1)$ have also been killed by Line 10. This implies that $\bar{p}_2 = \bar{q}_2 \in \mathbb{Z}_{\mathsf{per}(S)}$ is the only congruence class that is alive. Hence, the desired property is satisfied. $\square$

In later applications of Lemma 5.2.1, we will fix a randomly chosen $\sigma \in \{0,1\}^w$ with a suitable length $w = O(\log n)$. By a union bound, we can assume that $\mathsf{DS}_\sigma(S)$ is successful for all substrings S of the input string T.

5.3 Construction of Mapping ϕ

To build the τ-synchronizing set, our construction of the mapping ϕ will be an L-level procedure, from top to bottom with length parameters

$$\tau = \tau_L > \tau_{L-1} > \cdots > \tau_2 > \tau_1 = 6.$$

These parameters will be chosen later (we will choose $L = \Theta(\sqrt{\log \tau})$). In the following, we will apply Vishkin's deterministic sampling (Lemma 5.2.1) to substrings of the input string $T[1 \mathinner{..} n]$ of length $m \in \{\tau_1, \tau_2, \ldots, \tau_L\}$.

We first define a mapping π by augmenting the deterministic sample with some extra information.

Definition 5.3.1 (Mapping π)**.** For $S \in \Sigma^m$ with deterministic sample $\mathsf{DS}_\sigma(S) = (\delta; i_1, i_2, \ldots, i_w)$, define tuple

$$\pi_\sigma(S) := (\mathsf{per}(u); \delta; i_1, \ldots, i_w; S[i_1 - \delta], \ldots, S[i_w - \delta]).$$

Since $w = O(\log n)$, the tuple $\pi_\sigma(S)$ can be encoded using $\operatorname{poly}\log(n)$ bits. The following key lemma says π can distinguish two non-identical strings with large overlap.

Lemma 5.3.2. *Let $1 \le d \le \lfloor m/4 \rfloor$ and $S \in \Sigma^{m+d}$. Define two overlapping length-m substrings $U = S[1\mathinner{..}m], V = S[d+1\mathinner{..}d+m]$. Suppose seed σ generates successful deterministic samples for both U and V.*

Then, $\pi_\sigma(U) = \pi_\sigma(V)$ if and only if $U = V$, in which case d is a period of S.

Proof. The if part is immediate, and we shall prove the only if part.

Assuming $\pi_\sigma(U) = \pi_\sigma(V)$, let $\mathsf{per}(U) = \mathsf{per}(V) = p$, $\mathsf{DS}_\sigma(U) = \mathsf{DS}_\sigma(V) = (\delta; i_1, i_2, \ldots, i_w)$ where $0 \le \delta \le \lfloor m/2 \rfloor$ and $i_j - \delta \in [1\mathinner{..}m]$, and we have $U[i_j - \delta] = V[i_j - \delta]$ for all $j \in [w]$. We will first show d must be a multiple of p.

Suppose for contradiction that d is not a multiple of p. Since $\max\{\delta, \lfloor m/2 \rfloor - \delta\} \ge \lfloor m/4 \rfloor \ge d$, at least one of the following two cases must happen:

- If $d \le \delta$, then $0 \le \delta - d \le \lfloor m/2 \rfloor$, and the property of $\mathsf{DS}_\sigma(U)$ implies the existence of a checkpoint i_j with $i_j - (\delta - d) \in [1\mathinner{..}m]$ such that $U[i_j - \delta] \ne U[i_j - (\delta - d)]$, contradicting $U[i_j - \delta] = V[i_j - \delta] = U[i_j - \delta + d]$.

- If $d \le \lfloor m/2 \rfloor - \delta$, then $0 \le \delta + d \le \lfloor m/2 \rfloor$, and the property of $\mathsf{DS}_\sigma(V)$ implies the existence of a checkpoint i_j with $i_j - (\delta + d) \in [1\mathinner{..}m]$ such that $V[i_j - \delta] \ne V[i_j - (\delta + d)]$, contradicting $V[i_j - \delta] = U[i_j - \delta] = V[i_j - \delta - d]$.

Hence, d must be a multiple of p, and hence $p \le d \le \lfloor m/4 \rfloor < m - d$. Then, from $\mathsf{per}(S[1\mathinner{..}m]) = \mathsf{per}(S[d+1\mathinner{..}d+m]) = p < m - d$ we conclude $\mathsf{per}(S[1\mathinner{..}d+m]) = p$, and $U = S[1\mathinner{..}m] = S[d+1\mathinner{..}d+m] = V$. □

We define another auxiliary mapping ρ_ℓ for each layer $2 \le \ell \le L$, which will be useful for dealing with highly periodic substrings.

Definition 5.3.3 (Mapping ρ)**.** For $2 \le \ell \le L$ and $S \in \Sigma^{\tau_\ell}$, let

$$\rho_\ell(S) := \max\{r \in [1 \,..\, \tau_\ell] : \mathsf{per}(S[1 \,..\, r]) = \mathsf{per}(S[1 \,..\, \tau_{\ell-1}])\},$$

namely the furthest position that the period of the length-$\tau_{\ell-1}$ prefix can extend to.

We use Y_0 to denote $\{0,1\}^{\mathrm{poly}\log(n)}$, which can hold the return value of π_σ (a tuple) and the return value of ρ_ℓ (an integer). From now on we regard π_σ and ρ_ℓ as mappings from strings to Y_0.

Now we define the mapping ψ by concatenating the values returned by π and ρ in all layers.

Definition 5.3.4 (Mapping ψ)**.** Define mapping $\psi_\sigma \colon \Sigma^\tau \to Y_0^{2L-1}$ as follows: given $S[1 \,..\, \tau]$, let

$$\psi_\sigma(S) := p_1 r_2 p_2 r_3 p_3 \dots r_L p_L,$$

where

$$p_\ell = \pi_\sigma(S[1 \,..\, \tau_\ell]),\ 1 \le \ell \le L,$$

and

$$r_\ell = \rho_\ell(S[1 \,..\, \tau_\ell]),\ 2 \le \ell \le L.$$

Finally, we define a mapping $\phi \colon \Sigma^\tau \to Y^{2L-1}$ by simply hashing each symbol in $\psi(S)$, where $Y = [2^{\mathrm{poly}\log(n)}]$.

Definition 5.3.5 (Mapping ϕ)**.** Let seed $\sigma_{\mathsf{DS}} \in \{0,1\}^w$. Let another seed $\sigma_H \in \{0,1\}^{(2L-1)\cdot O(\log n)}$ independently sample $2L-1$ min-wise independent hash functions $h_1, h_2, \dots, h_{2L-1} \colon Y_0 \to Y$ from Lemma 2.7.1 with parameters n and $|Y_0|$.

Then, define mapping $\phi_{\sigma_{\mathsf{DS}},\sigma_H} \colon \Sigma^\tau \to Y^{2L-1}$ as follows: given $S \in \Sigma^\tau$ with

$$\psi_{\sigma_{\mathsf{DS}}}(S) = y_1 y_2 \cdots y_{2L-1},$$

define

$$\phi_{\sigma_{\mathsf{DS}},\sigma_H}(S) := h_1(y_1)h_2(y_2)\cdots h_{2L-1}(y_{2L-1}).$$

We sometimes omit the random seeds $\sigma, \sigma_{\mathsf{DS}}, \sigma_H$ from the subscripts and simply write $\psi(S), \phi(S)$.

We treat $\phi(S) \in Y^{2L-1}$ (and $\psi(S) \in Y_0^{2L-1}$) as length-$(2L-1)$ strings over alphabet Y (and Y_0). Elements in Y^{2L-1} are compared by their lexicographical order.

For $1 \le j \le 2L-1$, we also use $\phi(S)[1\,..\,j] \in Y^j$ to denote the length-j prefix of $\phi(S)$, and use $\phi(S)[j] \in Y$ to denote the j-th symbol of $\phi(S)$. Notations $\psi(S)[1\,..\,j], \psi(S)[j]$ are defined similarly.

5.4 Analysis of Sparsity

In this section, we analyze the sparsity of the synchronizing set A defined using mapping ϕ from Definition 5.3.5.

Recall that Q (defined in (5.1)) contains indices i with $\mathsf{per}(T[i\,..\,i+\tau)) \le \tau/3$, and recall that $\Phi(i) := \phi(T[i\,..\,i+\tau))$. We additionally define $\Psi(i) := \psi(T[i\,..\,i+\tau))$.

We have the following key lemma.

Lemma 5.4.1 (Expected count of minima). *Suppose interval $[l\,..\,r] \subseteq [1\,..\,n-\tau+1]$ has length $r-l+1 \le \tau/4$, and $[l\,..\,r] \cap \mathsf{Q} = \emptyset$.*

Then,

$$\mathop{\mathbf{E}}_{\sigma_{\mathsf{DS}},\sigma_H} \left|\left\{k \in [l\,..\,r] : \Phi(k) = \min\{\Phi(i) : i \in [l\,..\,k]\}\right\}\right| \le (O(\log \tau))^{2L-1},$$

and

$$\mathop{\mathbf{E}}_{\sigma_{\mathsf{DS}},\sigma_H} \left|\left\{k \in [l\,..\,r] : \Phi(k) = \min\{\Phi(i) : i \in [k\,..\,r]\}\right\}\right| \le (O(\log \tau))^{2L-1}.$$

Proof. We assume that the seed σ_{DS} is successful for all substrings of T. This happens with high probability over random $\sigma_{\mathsf{DS}} \in \{0,1\}^w$ with length $w = O(\log n)$.

First, we show that for $k_1, k_2 \in [l\,..\,r], k_1 \neq k_2$ we must have $\Psi(k_1) \neq \Psi(k_2)$. Suppose for contradiction that there are $k_1, k_2 \in [l\,..\,r], k_1 < k_2$ such that $\psi_{\sigma_{\mathsf{DS}}}(T[k_1\,..\,k_1 + \tau)) = \psi_{\sigma_{\mathsf{DS}}}(T[k_2\,..\,k_2 + \tau))$. Then, by definition of ψ (Definition 5.3.4), we have $\pi_{\sigma_{\mathsf{DS}}}(T[k_1\,..\,k_1+\tau)) = \pi_{\sigma_{\mathsf{DS}}}(T[k_2\,..\,k_2+\tau))$. By Lemma 5.3.2, since $k_2 - k_1 \leq \tau/4$, we must have $T[k_1\,..\,k_1+\tau) = T[k_2\,..\,k_2+\tau)$, and $(k_2 - k_1)$ is a period of $T[k_1\,..\,k_1+\tau)$. This implies $k_1 \in \mathsf{Q}$, contradicting $[l\,..\,r] \cap \mathsf{Q} = \emptyset$.

In the following, we will only prove the first expectation upper bound. The second statement can be proved similarly.

Recall that $\Phi(k) \in Y^{2L-1}$ is obtained by applying injective min-wise independent hash functions to $\Psi(k) \in Y_0^{2L-1}$, and they are compared in lexicographical order. In order for $\Phi(k)$ to be the lexicographically minimum among $\{\Phi(i)\}_{i\in[l..k]}$, its first symbol $\Phi(k)[1]$ needs to be the minimum among $\{\Phi(i)[1]\}_{i\in[l..k]}$, and then to break ties we compare the second symbols, and so on. Since the j-th symbol $\Phi(k)[j]$ is defined as the hash value $h_j(\Psi(k)[j])$, the probability of $\Phi(k)[j]$ being minimum at the j-th level is inversely proportional to the count of distinct $\Psi(i)[j]$ values that are being compared with.

Formally, for every $k \in [l\,..\,r]$ and every $1 \leq j \leq 2L-1$, define the count

$$c_{j,k} := \left|\{\Psi(k')[j] : k' \in [l\,..\,k], \Psi(k')[1\,..\,j-1] = \Psi(k)[1\,..\,j-1]\}\right|.$$

Then, conditioned on $\Phi(k)[1\,..\,j-1] = \min\{\Phi(k')[1\,..\,j-1] : k' \in [l\,..\,k]\}$, we have $\Phi(k)[1\,..\,j] = \min\{\Phi(k')[1\,..\,j] : k' \in [l\,..\,k]\}$ if and only if $\Phi(k)[j] = h_j(\Psi(k)[j])$ becomes the minimum among the $c_{j,k}$ candidates currently in a tie, which happens with at most $1.1/c_{j,k}$ probability by the min-wise independence of h_j. Hence, the probability that

$$\Phi(k) = \min\{\Phi(i) : i \in [l\,..\,k]\}$$

happens is at most

$$\frac{(1+0.1)^{2L-1}}{c_{1,k}\cdot c_{2,k}\cdots c_{2L-1,k}}. \tag{5.4}$$

We will derive the desired expectation upper bound by summing (5.4) over all $k \in$

$[l\,..\,r]$.

To do this, consider the following process of inserting strings $\Psi(l), \Psi(l+1), \ldots, \Psi(r) \in Y_0^{2L-1}$ one by one into a *trie*. Initially the trie only has a root node with node weight 1. Let $\Psi(k) \in Y_0^{2L-1}$ be the current string to be inserted. Starting from the root node, we iterate over $j = 1, 2, \ldots, 2L-1$, and each time move from the current node p down to its child node c labeled with symbol $\Psi(k)[j] \in Y_0$. If p does not have such a child c, then we have to first create a child node c labeled with $\Psi(k)[j]$, and assign c a real weight $1/\deg(p)$, where $\deg(p)$ denotes the number of children (including c) that p currently has. Now, observe that the value $1/\big(c_{1,k}c_{2,k}\ldots c_{2L-1,k}\big)$ is upper-bounded by the product of node weights on the path from the root node to the leaf representing $\Psi(k)$. This is because $c_{j,k}$ equals the number of children that the j-th node on this path currently has, which is not smaller than the reciprocals of the weights of its children (in particular, the $(j+1)$-st node on this path).

Each node in the trie has at most $r - l + 1 \le \lfloor \tau/4 \rfloor$ children, so the sum of the weights of its children is at most $1 + 1/2 + 1/3 + \cdots + 1/(\lfloor \tau/4 \rfloor) < \log \tau$. Since the trie has depth $2L-1$, a simple induction with distributive law of multiplication shows that the sum of node weight products over all root-to-leaf paths in the trie is at most $(\log \tau)^{2L-1}$.

We previously showed that $\Psi(k)$ are distinct for all $k \in [l\,..\,r]$, so each leaf in the trie corresponds to only one $k \in [l\,..\,r]$. Hence, the sum of $1/\big(c_{1,k}c_{2,k}\ldots c_{2L-1,k}\big)$ over all $k \in [l\,..\,r]$ is upper-bounded by $(\log \tau)^{2L-1}$. Then, the proof follows from summing over (5.4), which sums to at most $(1.1)^{2L-1} \cdot (\log \tau)^{2L-1} \le (O(\log \tau))^{2L-1}$. $\square$

Now we are ready to prove the sparsity of the synchronizing set A defined in (5.3).

Lemma 5.4.2 (Sparsity). *For every $i \in [1\,..\,n-3\tau+2]$, we have*

$$\mathop{\mathbf{E}}_{\sigma_{\mathsf{DS}},\sigma_H} \big|\mathsf{A} \cap [i\,..\,i+\tau)\big| \le (O(\log \tau))^{2L-1}.$$

Proof. Recall that for $k \in [i \mathinner{.\,.} i+\tau)$, we have $k \in \mathsf{A}$ if and only if

$$\min\{\Phi(j) : j \in [k \mathinner{.\,.} k+\tau] \setminus \mathsf{Q}\} \in \{\Phi(k), \Phi(k+\tau)\}.$$

To bound the number of such k, we will separately bound and sum up the number of $k \in [i \mathinner{.\,.} i+\tau)$ such that

$$\min\{\Phi(j) : j \in [k \mathinner{.\,.} k+\tau] \setminus \mathsf{Q}\} = \Phi(k), \tag{5.5}$$

and the number of $k' \in [i+\tau \mathinner{.\,.} i+2\tau)$ (where we replaced $k = k' - \tau$) such that

$$\min\{\Phi(j) : j \in [k'-\tau \mathinner{.\,.} k'] \setminus \mathsf{Q}\} = \Phi(k').$$

In the following, we will only bound the first case (the number of $k \in [i \mathinner{.\,.} i+\tau)$ satisfying (5.5)). The second case is symmetric and can be bounded similarly.

First we have the following claim.

Claim 5.4.3. *For all $k \in [i \mathinner{.\,.} i+\tau)$ satisfying (5.5), it holds that $k \notin \mathsf{Q}$.*

Proof of Claim 5.4.3. If $k \in \mathsf{Q}$, then there must exist $j \notin \mathsf{Q}$ such that $\Phi(j) = \Phi(k)$, which implies $\pi_{\sigma_{\mathsf{DS}}}(T[k \mathinner{.\,.} k+\tau]) = \pi_{\sigma_{\mathsf{DS}}}(T[j \mathinner{.\,.} j+\tau])$ by injectivity of the hash function h_L, and hence $\mathsf{per}(T[k \mathinner{.\,.} k+\tau]) = \mathsf{per}(T[j \mathinner{.\,.} j+\tau]) > \tau/3$, contradicting $k \in \mathsf{Q}$. $\square$

Hence, we only need to bound the number of $k \in [i \mathinner{.\,.} i+\tau) \setminus \mathsf{Q}$, which by Lemma 5.1.2 can be decomposed as the disjoint union of $O(1)$ many intervals, each having length at most $\tau/4$. For each such interval $[l' \mathinner{.\,.} r']$ (where $r' - l' + 1 \le \tau/4$ and $[l' \mathinner{.\,.} r'] \cap \mathsf{Q} = \emptyset$), any $k \in [l' \mathinner{.\,.} r']$ satisfying (5.5) must also satisfy $\min\{\Phi(j) : j \in [k \mathinner{.\,.} r']\} = \Phi(k)$. By Lemma 5.4.1, the expected number of such k in $[l' \mathinner{.\,.} r']$ is at most $(O(\log \tau))^{2L-1}$. Summing over $O(1)$ intervals $[l' \mathinner{.\,.} r']$, the total expected number of such k is also at most $(O(\log \tau))^{2L-1}$. $\square$

5.5 Algorithm for Reporting Synchronizing Positions

In this section we will give an efficient quantum algorithm for reporting the synchronizing positions in A.

First, observe that the mappings π (Definition 5.3.1) and ρ (Definition 5.3.3) can be computed efficiently.

Observation 5.5.1. *For $S \in \Sigma^m$ and seed $\sigma \in \{0,1\}^{O(\log n)}$, $\pi_\sigma(S)$ can be computed in $\widetilde{O}(\sqrt{m})$ quantum time.*

For $2 \le \ell \le L$ and $S \in \Sigma^{\tau_\ell}$, $\rho_\ell(S)$ can be computed in $\widetilde{O}(\sqrt{\tau_\ell})$ quantum time.

Hence, for all $1 \le \ell \le L, S \in \Sigma^{\tau_\ell}$, $\Phi(S)[1\mathinner{.\,.}2\ell-1]$ can be computed in $\widetilde{O}(\sqrt{\tau_\ell})$ quantum time.

Proof. The first statement immediately follows from Lemma 5.2.1 and Corollary 2.5.3. The second statement follows from Corollary 2.5.3 and a simple binary search with Grover search. The third statement follows by direct computing. □

We will need to efficiently identify elements in Q. The following lemma will be useful.

Lemma 5.5.2 (Finding a long cubic run)**.** *Given $S \in \Sigma^n$ with length $m \le n \le 4m/3$, there is a quantum algorithm in $\widetilde{O}(\sqrt{n})$ time that finds (if exists) a* maximal *substring $S[i\mathinner{.\,.}j]$ that has length $j-i+1 \ge m$ and period $\mathsf{per}(S[i\mathinner{.\,.}j]) \le m/3$, and such $S[i\mathinner{.\,.}j]$ is unique if exists.*

Proof. Since $j-i+1 \ge m$, $S[i\mathinner{.\,.}j]$ must contain $R := S[n-m+1\mathinner{.\,.}m]$ as a substring. Since $|R| = 2m-n \ge 2m/3 \ge 2\cdot\mathsf{per}(S[i\mathinner{.\,.}j])$, we must have $\mathsf{per}(S[i\mathinner{.\,.}j]) = \mathsf{per}(R)$. Hence, we can use Corollary 2.5.3 to compute $p = \mathsf{per}(R)$, and then uniquely determine i, j by extending the period to the left and to the right, namely,

$$i = \min\{i \in [1\mathinner{.\,.}n-m+1] : S[i'] = S[i'+p] \text{ for all } i' \in [i\mathinner{.\,.}n-m+1]\},$$
$$j = \max\{j \in [m\mathinner{.\,.}n] : S[j'] = S[j'-p] \text{ for all } j' \in [m\mathinner{.\,.}j]\},$$

using binary search with Grover search. □

Lemma 5.5.3 (Finding elements in Q)**.** *Given $i \in [1\,.\,.\,n-3\tau+2]$, one can compute $[i\,.\,.\,i+2\tau) \cap \mathsf{Q}$, represented as disjoint union of $O(1)$ intervals, in $\widetilde{O}(\sqrt{\tau})$ quantum time.*

Proof. By Lemma 5.1.2, the set $[i\,.\,.\,i+2\tau) \cap \mathsf{Q}$ can be decomposed into disjoint union of $O(1)$ many intervals. To find them, we separately consider overlapping blocks of length $\lfloor 4\tau/3 \rfloor$ inside $[i\,.\,.\,i+2\tau)$, with every adjacent two blocks being at a distance of $\lfloor \tau/6 \rfloor$. There are $O(1)$ many such blocks, and for every $j \in [i\,.\,.\,i+2\tau) \cap \mathsf{Q}$ there is some block that contains $[j\,.\,.\,j+\tau)$. Hence, we can apply Lemma 5.5.2 to each block, and find all j's (represented as an interval) such that $T[j\,.\,.\,j+\tau)$ is contained in this block and has period at most $\tau/3$. □

The main technical component is an algorithm that efficiently finds the minimal $\Phi(k)$ over an interval, stated in the following lemma.

Lemma 5.5.4 (Finding range minimum $\Phi(k)$)**.** *Given $[l\,.\,.\,r] \subseteq [1\,.\,.\,n-\tau+1]$ of length $r-l+1 \le \tau/4$, we can compute*

$$\arg\min_{k \in [l\,.\,.\,r]} \Phi(k),$$

represented as an arithmetic progression, in

$$\sqrt{\tau} \cdot (O(\log \tau))^L \cdot \operatorname{poly}\log(n) \cdot \sum_{\ell=1}^{L-1} \sqrt{\tau_{\ell+1}/\tau_\ell} \tag{5.6}$$

quantum time.

Before proving Lemma 5.5.4, we show that it can be used to report all the synchronizing positions in a length-τ interval, one at a time.

Lemma 5.5.5 (Efficient reporting)**.** *Given $i \in [1\,.\,.\,n-3\tau+2]$, the elements in $\mathsf{A} \cap [i\,.\,.\,i+\tau)$ can be reported in $O((\mathsf{cnt}+1)\cdot \mathcal{T})$ time, where $\mathsf{cnt} = |\mathsf{A} \cap [i\,.\,.\,i+\tau)|$ is the output count, and $\mathcal{T}$ is the time bound (5.6) in Lemma 5.5.4.*

Proof. To report $\mathsf{A} \cap [i \mathinner{.\,.} i+\tau)$, it suffices to report all $k \in [i \mathinner{.\,.} i+\tau)$ such that

$$\min\{\Phi(j) : j \in [k \mathinner{.\,.} k+\tau] \setminus \mathsf{Q}\} = \Phi(k), \tag{5.7}$$

and all $k' \in [i+\tau \mathinner{.\,.} i+2\tau)$ (where we replaced $k = k' - \tau$) such that

$$\min\{\Phi(j) : j \in [k'-\tau \mathinner{.\,.} k'] \setminus \mathsf{Q}\} = \Phi(k').$$

We will only show the first part. The second part is symmetric and can be solved similarly.

By Lemma 5.5.3, the set $[i \mathinner{.\,.} i+2\tau) \setminus \mathsf{Q}$ can be decomposed into disjoint union of $O(1)$ many intervals, which can be found in $\widetilde{O}(\sqrt{\tau})$ quantum time. Then, we can use Lemma 5.5.4 to answer $\arg\min\{\Phi(j) : j \in [l \mathinner{.\,.} r] \setminus \mathsf{Q}\}$ given any $[l \mathinner{.\,.} r] \subseteq [i \mathinner{.\,.} i+2\tau)$, in $O(\mathcal{T})$ time.

By Claim 5.4.3 (note that (5.7) is identical to (5.5)), it holds for all $k \in [i \mathinner{.\,.} i+\tau)$ satisfying (5.7) that $k \notin \mathsf{Q}$. We will report all $k \in [i \mathinner{.\,.} i+\tau) \setminus \mathsf{Q}$ satisfying (5.7) as follows:

Algorithm 6: Reporting all $k \in [i \mathinner{.\,.} i+\tau)$ satisfying (5.7)

```
1 begin
2     Let $l \leftarrow i$;
3     while $l < i+\tau$ do
4         Let $k$ be the minimum element in $\arg\min\{\Phi(j) : j \in [l \mathinner{.\,.} i+\tau) \setminus \mathsf{Q}\}$;
5         if $k$ satisfies (5.7) then
6             report $k$;
7             $l \leftarrow k+1$;
8         else
9             break
```

It is clear that the running time of Algorithm 6 is $O((\mathsf{cnt}+1) \cdot \mathcal{T})$, where cnt is the number of reported elements. It suffices to show that Algorithm 6 does not miss any k satisfying (5.7). We maintain the invariant that all not yet reported k satisfying (5.7) are at least l. Whenever we report k, it holds that $\Phi(k') > \Phi(k)$ for all $k' \in [l \mathinner{.\,.} k)$, so none of such k' can satisfy (5.7), and it is safe to skip them by letting $l \leftarrow k+1$. When the line of **break** is reached, we have $k \in \arg\min\{\Phi(j) : j \in [l \mathinner{.\,.} i+\tau) \setminus \mathsf{Q}\}$ but $k \notin \arg\min\{\Phi(j) : j \in [k \mathinner{.\,.} k+\tau) \setminus \mathsf{Q}\}$, implying that there exists $r \in [i+\tau \mathinner{.\,.} k+\tau)$ such that $\Phi(r) < \Phi(k')$ for all $k' \in [k \mathinner{.\,.} i+\tau)$, so none of $k' \in [k \mathinner{.\,.} i+\tau)$ can satisfy (5.7), and we can terminate the procedure. □

Now it remains to describe the algorithm in Lemma 5.5.4 for finding the minimum $\Phi(k)$ in an interval. Our algorithm is recursive, stated as follows.

Lemma 5.5.6. *For $1 \le \ell \le L$, there is a quantum algorithm running in $\mathcal{T}_j$ time that, given $[l \mathinner{.\,.} r] \subseteq [1 \mathinner{.\,.} n-\tau_\ell+1]$ of length $r-l+1 \le \tau_\ell/4$, computes*

$$\arg\min_{k\in[l\mathinner{.\,.}r]} \Phi(k)[1 \mathinner{.\,.} 2\ell-1]$$

represented as an arithmetic progression. The time complexities satisfy

$$\mathcal{T}_\ell \le \sqrt{\tau_\ell/\tau_{\ell-1}} \cdot \big(\mathcal{T}_{\ell-1} \cdot O(\log \tau) + \sqrt{\tau_\ell} \cdot \operatorname{poly}\log(n)\big) \tag{5.8}$$

for $\ell > 1$, and $\mathcal{T}_1 = \operatorname{poly}\log(n)$.

Proof. We first explain why the minimizers k form an arithmetic progression. For any $l \le k_1 < k_2 \le r$ with $\Phi(k_1)[1 \mathinner{.\,.} 2\ell-1] = \Phi(k_2)[1 \mathinner{.\,.} 2\ell-1]$, we have $\pi_{\sigma_{\mathsf{DS}}}(T[k_1 \mathinner{.\,.} k_1+\tau_\ell)) = \pi_{\sigma_{\mathsf{DS}}}(T[k_2 \mathinner{.\,.} k_2+\tau_\ell))$, and from Lemma 5.3.2 we have $T[k_1 \mathinner{.\,.} k_1+\tau_\ell) = T[k_2 \mathinner{.\,.} k_2+\tau_\ell)$ since $k_2 - k_1 \le \tau_\ell/4$. Hence, the minimizers k in this length-$\tau_\ell/4$ interval correspond to the starting positions of all occurrences of a certain length-τ_ℓ substring, which form an arithmetic progression by Lemma 2.1.3.

Due to lexicographical comparison rule, we know that any element in $\arg\min_{k\in[l\mathinner{.\,.}r]} \Phi(k)[1 \mathinner{.\,.} 2\ell-1]$ must also be in $\arg\min_{k\in[l\mathinner{.\,.}r]} \Phi(k)[1 \mathinner{.\,.} 2(\ell-1)-1]$, so we can recursively use the $(\ell-1)$-st level subroutine to find candidate minimizers. We first consider the simpler case where each $(\ell-1)$-st level subroutine only returns a single minimizer. Given the input interval $[l \mathinner{.\,.} r]$ of length $r-l+1 \le \tau_\ell/4$, we divide $[l \mathinner{.\,.} r]$ into blocks each of length $\tau_{\ell-1}/4$, on which we recursively apply the $(\ell-1)$-st level subroutine (boosted to $1-1/\operatorname{poly}(\tau)$ success probability by $O(\log \tau)$ repetitions) to find the minimizer k' in this block, and then use $\sqrt{\tau_\ell}\cdot\mathrm{poly}\log(n)$ time to compute $\Phi(k')[1\,..\,2\ell-1]$. Out of the $\tau_\ell/\tau_{\ell-1}$ blocks, we apply quantum minimum finding to find the block that has the minimum $\Phi(k')[1\,..\,2\ell-1]$ (Observation 5.5.1). This quantum minimum finding procedure incurs $O(\sqrt{\tau_\ell/\tau_{\ell-1}})$ comparisons, and results in the time bound (5.8).

Now we consider the general case, where for a $(\ell-1)$-st level block $[l'\,..\,r']$, $\arg\min_{k\in[l'\,..\,r']}\Phi(k)[1\,..\,2(\ell-1)-1]$ may contain multiple elements $k_1 < k_2 < \cdots < k_d$ forming an arithmetic progression, where $k_d - k_1 \le \tau_{\ell-1}/4$. We already know from Lemma 2.1.3 that $T[k_j\,..\,k_j+\tau_{\ell-1})$ are identical for all $j\in[d]$, and $p = k_2-k_1 = \cdots = k_d - k_{d-1} = \mathsf{per}(T[k_1\,..\,k_d+\tau_{\ell-1}))$. To find the minimum $\Phi(k)[1\,..\,2\ell-1]$ among $k\in\{k_1,\ldots,k_d\}$, we will additionally compare $\Phi(k)[2(\ell-1)] = h_{2(\ell-1)}\big(\rho_\ell(T[k\,..\,k+\tau_\ell))\big)$ to break ties.

Now, let $r' = \max\{r' : r' \le k_d+\tau_\ell, \mathsf{per}(T[k_1\,..\,r')) = p\}$, namely the furthest point this period can extend to (up to a distance of the current level interval length τ_ℓ). Now we can easily obtain their ρ_ℓ values,

$$\rho_\ell(T[k_j\,..\,k_j+\tau_\ell)) = \min\{\tau_\ell, r'-k_j\}.$$

Then, among strings $T[k_j\,..\,k_j+\tau_{\ell-1})$, $j\in[d]$, those with ρ_ℓ value equal to τ_ℓ are all identical to $T[k_1\,..\,k_1+\tau)$, and the remaining ones must have different ρ_ℓ values that are strictly smaller than τ_ℓ. Hence, we can simply use Grover search over $j\in[d]$ in $\widetilde{O}(\sqrt{d}) \le \widetilde{O}(\sqrt{\tau_{\ell-1}})$ time to find the minimum $\Phi(k_j)[2(\ell-1)] = h_{2(\ell-1)}\big(\rho_\ell(T[k_j\,..\,k_j+\tau_\ell))\big)$. $\square$

Proof of Lemma 5.5.4 using Lemma 5.5.6. Note that the algorithm in Lemma 5.5.6 with $\ell = L$ solves the task required by Lemma 5.5.4. The quantum time complexity $\mathcal{T}_L$ can be calculated by expanding (5.8) into

$$\mathcal{T}_L \le \sqrt{\tau_L}\sqrt{\frac{\tau_L}{\tau_{L-1}}}\,\mathrm{poly}\log(n) + \sqrt{\frac{\tau_L}{\tau_{L-1}}}\mathcal{T}_{L-1}\cdot O(\log\tau)$$

$$
\begin{aligned}
&\le \sqrt{\tau_L}\sqrt{\frac{\tau_L}{\tau_{L-1}}}\operatorname{poly}\log(n) + \sqrt{\tau_L}\sqrt{\frac{\tau_{L-1}}{\tau_{L-2}}}\operatorname{poly}\log(n)\cdot O(\log\tau) + \sqrt{\frac{\tau_L}{\tau_{L-2}}}\mathcal{T}_{L-2}(O(\log\tau))^2 \\
&\le \cdots \\
&\le \sqrt{\tau_L}\sum_{\ell=1}^{L-1}\sqrt{\frac{\tau_{\ell+1}}{\tau_\ell}}\operatorname{poly}\log(n)\cdot(O(\log\tau))^{L-\ell-1} \qquad \square
\end{aligned}
$$

Finally, we prove the main theorem by setting the length parameters $\tau_1, \ldots, \tau_L$ appropriately.

Proof of Theorem 1.2.5. We set $L = \Theta(\sqrt{\log\tau})$, and pick $6 \le \tau_1 < \tau_2 < \cdots < \tau_L = \tau$ so that $\tau_{\ell+1}/\tau_\ell \le O(\tau^{1/L})$ for all $1 \le \ell < L$. Then, by (5.6), the quantum time complexity of the range minimum subroutine (Lemma 5.5.4) is at most

$$
\begin{aligned}
&\sqrt{\tau}\cdot(O(\log\tau))^L\cdot\operatorname{poly}\log(n)\cdot L\cdot O(\tau^{1/2L}) \\
\le\ &\sqrt{\tau}\cdot\operatorname{poly}\log(n)\cdot(O(\log\tau))^{O(\sqrt{\log\tau})} \\
\le\ &\tau^{1/2+o(1)}\cdot\operatorname{poly}\log(n).
\end{aligned}
$$

Recall that with high probability $\sigma_{\mathsf{DS}} \in \{0,1\}^{O(\log n)}$ succeeds for all substrings S of T. The rest of the proof follows from Lemma 5.1.1, Lemma 5.4.2, and Lemma 5.5.5. $\square$

Chapter 6

Conclusion

We conclude by mentioning several open questions related to our work.

- In our quantum string synchronizing set result (Theorem 1.2.5), can we improve the extra $\tau^{o(1)}$ factors in the sparsity and the time complexity of to polylogarithmic?

- String synchronizing sets have had many applications in classical string algorithms [63, 6, 40, 64, 65, 66]. Can our new result find more applications in quantum string algorithms?

- Our $\widetilde{O}(n^{2/3})$-time algorithm for LCS assumes that the input characters are integers in $[\mathrm{poly}(n)]$. This assumption was crucially used in the string synchronizing set algorithm (Chapter 5). However, the previous $\widetilde{O}(n^{5/6})$-time LCS algorithm by Le Gall and Seddighin [51] can work with *general ordered alphabet*, where the only allowed query is to compare two symbols $S[i], S[j]$ in the input strings (with three possible outcomes $S[i] > S[j], S[i] = S[j]$, or $S[i] < S[j]$). Is $\widetilde{O}(n^{2/3})$ query complexity (or even time complexity) achievable in this more restricted setting? Alternatively, can we show a better query lower bound?

Bibliography

[1] Scott Aaronson, Nai-Hui Chia, Han-Hsuan Lin, Chunhao Wang, and Ruizhe Zhang. On the quantum complexity of closest pair and related problems. In *Proceedings of the 35th Computational Complexity Conference (CCC 2020)*, pages 16:1–16:43, 2020. 23, 30

[2] Scott Aaronson, Daniel Grier, and Luke Schaeffer. A quantum query complexity trichotomy for regular languages. In *Proceedings of the 60th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2019)*, pages 942–965, 2019. 10

[3] Scott Aaronson and Yaoyun Shi. Quantum lower bounds for the collision and the element distinctness problems. *J. ACM*, 51(4):595–605, 2004. 10, 12, 57

[4] Amir Abboud, Richard Ryan Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2015)*, pages 218–230, 2015. 24

[5] Shyan Akmal and Ce Jin. Near-optimal quantum algorithms for string problems. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2791–2832, 2022. 5, 15

[6] Mai Alzamel, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba. Quasi-linear-time algorithm for longest common circular factor. In *Proceedings of the 30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*, pages 25:1–25:14, 2019. 14, 87

[7] Andris Ambainis. Quantum query algorithms and lower bounds. In *Classical and New Paradigms of Computation and their Complexity Hierarchies*, pages 15–32. Springer, 2004. 30

[8] Andris Ambainis. Polynomial degree and lower bounds in quantum complexity: Collision and element distinctness with small range. *Theory Comput.*, 1(1):37–46, 2005. 10, 57

[9] Andris Ambainis. Quantum walk algorithm for element distinctness. *SIAM J. Comput.*, 37(1):210–239, 2007. 10, 12, 15, 23, 30, 32, 33, 35, 42, 43, 45, 46, 48, 49

[10] 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 *Proceedings of the 45th International Symposium on Mathematical Foundations of Computer Science (MFCS 2020)*, pages 8:1–8:14, 2020. 10, 23

[11] Andris Ambainis and Nikita Larka. Quantum algorithms for computational geometry problems. In *Proceedings of the 15th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2020)*, pages 9:1–9:10, 2020. 24

[12] Andris Ambainis and Ashley Montanaro. Quantum algorithms for search with wildcards and combinatorial group testing. *Quantum Inf. Comput.*, 14(5-6):439–453, 2014. 23

[13] Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common substring made fully dynamic. In *Proceedings of the 27th Annual European Symposium on Algorithms (ESA 2019)*, pages 6:1–6:17, 2019. 16, 24, 36

[14] Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. 16, 24, 36

[15] Alberto Apostolico and Maxime Crochemore. Optimal canonization of all substrings of a string. *Inf. Comput.*, 95(1):76–95, 1991. 24

[16] Alberto Apostolico, Costas S. Iliopoulos, and Robert Paige. On O(n log n) cost parallel algorithm for the single function coarsest partition problem. In *Parallel Algorithms and Architectures, International Workshop, 1987, Proceedings*, pages 70–76, 1987. 10

[17] Maxim A. Babenko, Paweł Gawrychowski, Tomasz Kociumaka, Ignat I. Kolesnichenko, and Tatiana Starikovskaya. Computing minimal and maximal suffixes of a substring. *Theor. Comput. Sci.*, 638:112–121, 2016. 24

[18] Maxim A. Babenko, Ignat I. Kolesnichenko, and Tatiana Starikovskaya. On minimal and maximal suffixes of a substring. In *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*, pages 28–37. Springer, 2013. 24

[19] Maxim A. Babenko and Tatiana Starikovskaya. Computing longest common substrings via suffix arrays. In *Proceedings of the 3rd International Computer Science Symposium in Russia (CSR 2008), Theory and Applications*, pages 64–75, 2008. 9, 24

[20] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.*, 47(3):1087–1097, 2018. 22

[21] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Phys. Rev. A*, 52:3457–3467, 1995. 30

[22] Aleksandrs Belovs, Andrew M. Childs, Stacey Jeffery, Robin Kothari, and Frédéric Magniez. Time-efficient quantum walks for 3-distinctness. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP 2013), Part I*, pages 105–122, 2013. 23

[23] Stav Ben-Nun, Shay Golan, Tomasz Kociumaka, and Matan Kraus. Time-space tradeoffs for finding a long common substring. In *Proceedings of the 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, pages 5:1–5:14, 2020. 16, 24, 36

[24] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh V. Vazirani. Strengths and weaknesses of quantum computing. *SIAM J. Comput.*, 26(5):1510–1523, 1997. 10, 12

[25] Daniel J. Bernstein, Stacey Jeffery, Tanja Lange, and Alexander Meurer. Quantum algorithms for the subset-sum problem. In *Proceedings of the 5th International Workshop on Post-Quantum Cryptography (PQCrypto 2013)*, pages 16–33, 2013. 23

[26] Philip Bille, Paweł Gawrychowski, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. Longest common extensions in trees. In *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, pages 52–64, 2015. 36

[27] Guy E. Blelloch, Daniel Golovin, and Virginia Vassilevska. Uniquely represented data structures for computational geometry. In *Proceedings of the 11th Scandinavian Workshop on Algorithm Theory (SWAT 2008)*, pages 17–28, 2008. 44

[28] Kellogg S. Booth. Lexicographically least circular substrings. *Inf. Process. Lett.*, 10(4/5):240–242, 1980. 9, 24

[29] Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, MohammadTaghi Haji-Aghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and MapReduce. *J. ACM*, 68(3):1–41, 2021. 10, 22

[30] Gilles Brassard, Peter Høyer, Kassem Kalach, Marc Kaplan, Sophie Laplante, and Louis Salvail. Key establishment à la merkle in a quantum world. *J. Cryptol.*, 32(3):601–634, 2019. 57

[31] Gilles Brassard, Peter Høyer, Michele Mosca, and Alain Tapp. Quantum amplitude amplification and estimation. *arXiv preprint quant-ph/0005055*, 2000. 31

[32] Harry Buhrman and Ronald de Wolf. Complexity measures and decision tree complexity: a survey. *Theor. Comput. Sci.*, 288(1):21–43, 2002. 30

[33] Harry Buhrman, Bruno Loff, Subhasree Patro, and Florian Speelman. Limits of quantum speed-ups for computational geometry and other problems: Fine-grained complexity via quantum walks. In *Proceedings of the 13th Innovations in Theoretical Computer Science Conference (ITCS 2022)*, pages 31:1–31:12, 2022. 24, 43, 46, 49

[34] Harry Buhrman, Subhasree Patro, and Florian Speelman. A framework of quantum strong exponential-time hypotheses. In *Proceedings of the 38th International Symposium on Theoretical Aspects of Computer Science (STACS 2021)*, pages 19:1–19:19, 2021. 10, 24

[35] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, pages 55–69, 2003. 36

[36] Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucký, and Michael E. Saks. Approximating edit distance within constant factor in truly sub-quadratic time. *J. ACM*, 67(6):36:1–36:22, 2020. 22

[37] Timothy M. Chan and Konstantinos Tsakalidis. Dynamic orthogonal range searching on the RAM, revisited. In *Proceedings of the 33rd International Symposium on Computational Geometry (SoCG 2017)*, pages 28:1–28:13, 2017. 18

[38] Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Linear-time algorithm for long LCF with k mismatches. In *Proceedings of the 29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*, pages 23:1–23:16, 2018. 16, 17, 24, 36, 39, 43

[39] Panagiotis Charalampopoulos, Paweł Gawrychowski, and Karol Pokorski. Dynamic longest common substring in polylogarithmic time. In *Proceedings of the 47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*, pages 27:1–27:19, 2020. 16, 17, 18, 24, 36, 43, 44

[40] Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Faster algorithms for longest common substring. In *Proceedings of the 29th Annual European Symposium on Algorithms (ESA 2021)*, pages 30:1–30:17, 2021. 14, 16, 24, 36, 54, 55, 87

[41] Richard Cleve, Kazuo Iwama, François Le Gall, Harumichi Nishimura, Seiichiro Tani, Junichi Teruyama, and Shigeru Yamashita. Reconstructing strings from substrings with quantum queries. In *Proceedings of the 13th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2012)*, pages 388–397, 2012. 23

[42] Maxime Crochemore, Leszek Gasieniec, Wojciech Plandowski, and Wojciech Rytter. Two-dimensional pattern matching in linear time and small space. In *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1995)*, pages 181–192, 1995. 21

[43] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. 24

[44] Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology*. World Scientific, 2002. 24

[45] Ronald de Wolf. Quantum computing: Lecture notes. *CoRR*, abs/1907.09415v2, 2019. 33

[46] Christoph Durr and Peter Høyer. A quantum algorithm for finding the minimum. *arXiv preprint quant-ph/9607014*, 1996. 10, 21, 31

[47] Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983. 9, 24

[48] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS 1997)*, pages 137–143, 1997. 9, 24

[49] Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. 28

[50] Tomáš Flouri, Emanuele Giaquinta, Kassian Kobert, and Esko Ukkonen. Longest common substrings with k mismatches. *Inf. Process. Lett.*, 115(6-8):643–647, 2015. 24

[51] François Le Gall and Saeed Seddighin. Quantum meets fine-grained complexity: Sublinear time quantum algorithms for string problems. In *Proceedings of the 13th Innovations in Theoretical Computer Science Conference (ITCS 2022)*, volume 215, pages 97:1–97:23, 2022. 9, 10, 11, 15, 23, 30, 35, 36, 58, 87

[52] Leszek Gasieniec, Wojciech Plandowski, and Wojciech Rytter. Constant-space string matching with smaller number of comparisons: Sequential sampling. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM 1995)*, pages 78–89, 1995. 21

[53] Paweł Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Waleń. Faster longest common extension queries in strings over general alphabets. In *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, pages 5:1–5:13, 2016. 36

[54] Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya. Approximating longest common substring with k mismatches: Theory and practice. In *Proceedings of the 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, pages 16:1–16:15, 2020. 24

[55] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC 1996)*, pages 212–219, 1996. 9, 12, 31

[56] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. 24

[57] Costas S. Iliopoulos and William F. Smyth. Optimal algorithms for computing the canonical form of a circular string. *Theor. Comput. Sci.*, 92(1):87–105, 1992. 10, 24, 61

[58] Piotr Indyk. A small approximately min-wise independent family of hash functions. *J. Algorithms*, 38(1):84–90, 2001. 33

[59] Stacey Jeffery. *Frameworks for quantum algorithms*. PhD thesis, University of Waterloo, 2014. 23

[60] Stacey Jeffery, Robin Kothari, and Frédéric Magniez. Nested quantum walks with quantum data structures. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2013)*, pages 1474–1485, 2013. 23

[61] Ce Jin and Jakob Nogler. Quantum speed-ups for string synchronizing sets, longest common substring, and k-mismatch matching, 2022. Unpublished manuscript. 5, 15

[62] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987. 9

[63] Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC 2019)*, pages 756–767. ACM, 2019. 13, 14, 16, 20, 69, 70, 87

[64] Dominik Kempa and Tomasz Kociumaka. Resolution of the burrows-wheeler transform conjecture. In Sandy Irani, editor, *Proceedings of the 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS 2020)*, pages 1002–1013, 2020. 14, 87

[65] Dominik Kempa and Tomasz Kociumaka. Breaking the O(n)-barrier in the construction of compressed suffix arrays. *CoRR*, abs/2106.12725, 2021. 14, 87

[66] Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2022)*, pages 1657–1670, 2022. 14, 87

[67] Carmel Kent, Moshe Lewenstein, and Dafna Sheinwald. On demand string sorting over unbounded alphabets. *Theor. Comput. Sci.*, 426:66–74, 2012. 28

[68] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. 9, 64

[69] Tomasz Kociumaka. Minimal suffix and rotation of a substring in optimal time. In *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, pages 28:1–28:12, 2016. 24

[70] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2015)*, pages 532–551, 2015. 28, 32

[71] Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya. Longest common substring with approximately k mismatches. *Algorithmica*, 81(6):2633–2652, 2019. 24

[72] Tomasz Kociumaka, Tatiana Starikovskaya, and Hjalte Wedel Vildhøj. Sublinear space algorithms for the longest common substring problem. In *Proceedings of the 22th Annual European Symposium on Algorithms (ESA 2014)*, pages 605–617, 2014. 24

[73] Samuel Kutin. Quantum lower bound for the collision problem with small range. *Theory Comput.*, 1(1):29–36, 2005. 10

[74] François Le Gall. Improved quantum algorithm for triangle finding via combinatorial arguments. In *Proceedings of the 55th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2014)*, pages 216–225, 2014. 23

[75] Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985. 36

[76] Frédéric Magniez, Ashwin Nayak, Jérémie Roland, and Miklos Santha. Search via quantum walk. *SIAM J. Comput.*, 40(1):142–164, 2011. 17, 23, 32, 33

[77] Frédéric Magniez, Miklos Santha, and Mario Szegedy. Quantum algorithms for the triangle problem. *SIAM J. Comput.*, 37(2):413–424, 2007. 23

[78] William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980. 22

[79] Ashley Montanaro. Quantum pattern matching fast on average. *Algorithmica*, 77(1):16–39, 2017. 23

[80] Christian Worm Mortensen. Fully dynamic orthogonal range reporting on RAM. *SIAM J. Comput.*, 35(6):1494–1525, 2006. 18

[81] Timothy Naumovitz, Michael E. Saks, and C. Seshadhri. Accurate and nearly optimal sublinear approximations to ulam distance. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, pages 2012–2031, 2017. 23

[82] Wojciech Plandowski and Wojciech Rytter. Application of Lempel-Ziv encodings to the solution of words equations. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP 1998)*, pages 731–742, 1998. 28

[83] William Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, University of Maryland at College Park, USA, 1990. 47

[84] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990. 45

[85] H. Ramesh and V. Vinay. String matching in $\widetilde{O}(\sqrt{n}+\sqrt{m})$ quantum time. *J. Discrete Algorithms*, 1(1):103–110, 2003. 9, 10, 19, 21, 22, 31, 32, 71

[86] Yossi Shiloach. Fast canonization of circular strings. *J. Algorithms*, 2(2):107–121, 1981. 9, 24

[87] Tatiana Starikovskaya. Longest common substring with approximately k mismatches. In *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, pages 21:1–21:11, 2016. 24

[88] Tatiana Starikovskaya and Hjalte Wedel Vildhøj. Time-space trade-offs for the longest common substring problem. In *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*, pages 223–234, 2013. 16, 24, 36

[89] Mario Szegedy. Quantum speed-up of Markov chain based algorithms. In *Proceedings of the 45th Symposium on Foundations of Computer Science (FOCS 2004)*, pages 32–41, 2004. 23, 32

[90] Sharma V. Thankachan, Alberto Apostolico, and Srinivas Aluru. A provably efficient algorithm for the k-mismatch average common substring problem. *J. Comput. Biol.*, 23(6):472–482, 2016. 24

[91] Uzi Vishkin. Deterministic sampling - A new technique for fast pattern matching. *SIAM J. Comput.*, 20(1):22–40, 1991. 9, 10, 21, 31, 61, 71

[92] Qisheng Wang and Mingsheng Ying. Quantum algorithm for lexicographically minimal string rotation. *CoRR*, abs/2012.09376, 2020. 10, 11, 13, 19, 22, 61, 62, 71

[93] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973. 9, 10, 24

[94] Dan E. Willard and George S. Lueker. Adding range restriction capability to dynamic data structures. *J. ACM*, 32(3):597–617, 1985. 18