STRING MEASURES: COMPUTATIONAL COMPLEXITY AND RELATED PROBLEMS IN COMMUNICATION

by
Yu Zheng

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy

Baltimore, Maryland
May, 2022

Abstract

Strings are fundamental objects in computer science. Modern applications such as text processing, bioinformatics, and distributed data storage systems often need to deal with very large strings. These applications motivated the study of the computational complexity of string related problems as well as a better understanding of edit operations on strings in general. In this thesis, we study several problems related to edit type string measures and error correcting codes for edit errors, i.e. insertions and deletions.

The results presented in this thesis can be roughly partitioned into two parts. The first part is about the space complexity of computing or approximating string measures. We study three classical string measures: edit distance (ED), longest common subsequence (LCS), and longest increasing subsequence (LIS). Our first main result shows that all these three string measures can be approximated to within a $1 + o(1)$ multiplicative factor using only polylog space in polynomial time. We further study ED and LCS in the asymmetric streaming model introduced by Saks and Seshadhri (SODA, 2013). The model can be viewed as an intermediate model between the random access model and the standard streaming model. In this model, one has streaming access to one of the input strings and random access to the other. For both ED and LCS, we present new algorithms as well as several space lower bounds in the asymmetric streaming model.

The second part of our results is about locally decodable codes (LDCs) that can tolerate edit errors. LDCs are a class of error correcting code that allow quick recovery of a message symbol by only looking at a few positions of the encoded message (codeword). LDCs for Hamming errors have been extensively studied while arguably little is known about LDCs for edit errors. In this thesis, we present exponential lower bounds for LDCs that can tolerate edit errors. In particular, we show that 2-query linear LDCs for edit errors do not exist, and the codeword length of any constant query LDCs for edit errors must be exponential. These bounds exhibit a strict separation between Hamming errors and edit errors. We also introduce the notion of LDCs with randomized encoding, which can be viewed as a relaxation of the standard LDCs. We give constructions of LDCs with randomized encoding that can achieve significantly better rate-query tradeoff.

Thesis Readers

Dr. Xin Li (Primary Advisor)
Associate Professor
Department of Computer Science
Johns Hopkins University

Dr. Amitabh Basu
Associate Professor
Department of Applied Mathematics and Statistics
Johns Hopkins University

Dr. Michael Dinitz
Associate Professor
Department of Computer Science
Johns Hopkins University

Dedicated to my family.

Acknowledgements

First and foremost, I am grateful to my advisor, Xin Li. Xin is a supportive advisor and helped me throughout my PhD study. He taught me about theoretical computer science research and introduced me to the problems that eventually formed this thesis. Xin is a dedicated researcher full of great ideas and insightful intuitions about our studied problems. I have learned a lot from numerous discussions with him. He also encouraged me during the hard times of my study. I would not be able to finish this thesis without his encouragement and support.

In the past five years, I am fortunate to have had opportunities to work with many researchers. These collaborations led to the results presented in this thesis. I am especially thankful to Alexander Block, Jeremiah Blocki, Kuan Cheng, Zhengzhong Jin, Ventaka Gandikota, Elena Grigorescu, and Minshen Zhu. It is a great joy to work with them. I have learned from them in various ways.

I am grateful to my thesis committee and my GBO committee: Amitabh Basu, Michael Dinitz, Edinah Gnang, and Vladimir Braverman. They examined my research works and provided many useful suggestions.

I would also like to thank many other professors at Johns Hopkins who taught me various topics in computer science and discussed research problems with me. During my PhD study, I got to know many fellow graduate students. I want to thank them for the useful advice and for making my life at Johns Hopkins much more enjoyable.

Finally, I am grateful to my family. My father Jie Zheng inspired me to pursue what

I am interested in and my mother Jianxiang Wu is always supportive and encouraging. I am fortunate to have such great parents. I also want to thank my fiancée, Yingchao Xue, for her love, care, and support. Words are not enough to describe my gratitude to them.

Contents

List of Algorithms

Chapter 1

Introduction

Strings are fundamental objects in computer science with a variety of applications. Many interesting computational problems arise from the analysis of *edit operations* (insertions, deletions, and substitutions) on strings. For example, the edit distance between two strings is the minimum number of edit operations that are required to transform one string to the other. It is a useful measure for the similarity between two strings. Modern applications often need to compute or approximate edit distance between very large strings. The classical dynamic programming algorithm becomes infeasible in many circumstances. Thus, designing algorithms with improved computational complexity (either time or space complexity) has drawn attention from the theoretical computer science research community. Many breakthrough results about computing or approximating string similarity measures have been achieved during the past decade. Another motivation for studying the edit operations is from the research of error correcting codes (ECCs) that can tolerate edit errors, i.e. insertions and deletions. Inspired by edit errors in communication and the study of DNA storage (e.g. [1]), ECCs for edit errors have been studied in many recent works. We believe a better understanding of edit operations will lead to new results about ECCs for edit errors.

Remark 1.1. *In this thesis, we define the edit distance as the minimum number of* *insertions, deletions, and substitutions that are required to transform one string to another. But when talking about ECCs, we define edit errors to be insertions and deletions, not including substitutions. We note that each substitution can be replaced by one insertion and one deletion. Thus whether or not including substitutions does not affect the nature of edit errors. We define edit errors in this way to make the notations consistent with the convention in the literature.*

In this thesis, we study several problems related to edit operations. We now give an introduction to these problems.

1.1 Approximation with Small Space

The first problem we study in this thesis is the space complexity of approximating the following three classical string measures:

Edit distance (or Levenshtein distance): given two strings, the edit distance (ED) between these strings is the minimum number of insertions, deletions, and substitutions to transform one string into another.

Longest common subsequence: given two strings, the longest common subsequence (LCS) between these strings is the longest subsequence that appears in both strings.

Longest increasing subsequence: given one string and a total order over the alphabet, the longest increasing subsequence (LIS) is the longest sequence in the string that is in an increasing order.

These problems have found applications in a wide range of applications, including bioinformatics, text processing, compilers, data analysis and so on. As a result, all of them have been studied extensively. Specifically, suppose the length of each string is n, then both ED and LCS can be computed in $O(n^2)$ time and $O(n \log n)$ bits of space using standard dynamic programming. For LIS, it is known that it can be computed exactly in time $O(n \log n)$ with $O(n \log n)$ bits of space. However, in practical applications these problems often arise in situations of huge data sets, where the magnitude of n can be in the order of billions (for example, when one studies human gene sequences). Thus, even a running time of $\Theta(n^2)$ can be too costly. Similarly, even a $\Theta(n)$ memory consumption can be infeasible in many applications, especially for basic tasks such as ED, LCS, and LIS since they are often used as building blocks of more complicated algorithms.

Motivated by this, there have been many attempts at reducing the time complexity of computing ED and LCS, however none of these attempts succeeded significantly. Recent advances in fine grained complexity provide a justification for these failures, where the work of Backurs and Indyk [2] and the work of Abboud, Backurs, and Williams [3] show that no algorithm can compute ED or LCS in time $O(n^{1.99})$ unless the Strong Exponential Time Hypothesis (SETH) [4] is false. Since then, the focus has been on developing *approximation* algorithms for ED and LCS with significantly better running time, and there has been much success here. In particular, following a recent breakthrough result [5], which gives the first constant factor approximation of ED in truly sub-quadratic time, subsequent improvements have finally achieved a constant factor approximation of ED in near linear time [6–8]. For LCS the situation appears to be harder, and the best known randomized algorithm [9] only achieves an $O(n^{0.498})$ approximation using linear time, which slightly beats the trivial $O(\sqrt{n})$ approximation obtained by sampling. Additionally, there is a trivial linear time algorithm that can approximate LCS within a factor of $1/|\Sigma|$ where Σ is the alphabet of the strings. A recent work [10] further provides a randomized algorithm in truly sub-quadratic time that achieves an approximation factor of $O(\lambda^3)$, where λ is the ratio of the optimal solution size over the input size. Another recent work by Rubinstein and Song [11] shows how to reduce LCS to ED for binary strings, and uses the reduction to give a near linear time $\frac{1}{2} + \varepsilon$ approximation algorithm for LCS of binary strings, where $\varepsilon > 0$ is some constant.

Despite these success, the equally important question of approximating ED and LCS using small space has not been studied in depth. Only a few previous works have touched on this topic, but with a different focus. For example, assume the edit distance between two strings is at most k, the work of Chakraborty et. al. [12] provides a randomized streaming algorithm that obtains an $O(k)$ approximation of ED, using linear time and $O(\log n)$ space. Based on this, the work of Belazzougui and Zhang [13] provides a randomized streaming algorithm for computing ED and LCS *exactly* using polynomial time and $\mathsf{poly}(k \log n)$ space.

For LIS the situation is slightly better. In particular, the work of Gopalan et. al. [14] provides a deterministic streaming algorithm that approximates LIS to within a $1 - \varepsilon$ factor, using time $O(n \log n)$ and space $O(\sqrt{n/\varepsilon} \log n)$; while the work by Kiyomi et. al. [15] obtains a deterministic algorithm that computes LIS *exactly* using $O(n^{1.5} \log n)$ time and $O(\sqrt{n} \log n)$ space.

We seek to better understand the space complexity of these problems, while at the same time maintaining a polynomial running time. The first and most natural goal would be to see if we can compute for example ED and LCS exactly using significantly smaller space (i.e., truly sub-linear space of $n^{1-\alpha}$ for some constant $\alpha > 0$). However, this again appears to be hard as no success has been achieved in the literature so far. Thus, we turn to a more realistic goal — to approximate ED and LCS using significantly smaller space. For LIS, our goal is to use approximation to further reduce the space complexity in [14] and [15].

More broadly, these questions are closely related to the general question of *non-deterministic small space computation* vs. *deterministic small space computation*. Specifically, the decision versions of all three problems (ED, LCS, LIS) can be easily shown to be in the class NL (i.e., non-deterministic log-space), and the question of whether $\mathsf{NL} = \mathsf{L}$ (i.e., if non-deterministic log-space computation is equivalent to deterministic log-space computation) is a major open question in complexity theory. Note that if $\mathsf{NL} = \mathsf{L}$, this would trivially imply polynomial time algorithms for *exactly* computing ED, LCS, and LIS in logspace. However, although we know that $\mathsf{NL} \subseteq \mathsf{P}$ and $\mathsf{NL} \subseteq \mathsf{SPACE}(\log^2 n)$ (by Savitch's theorem [16]), it is not known if every problem in NL can be solved simultaneously in polynomial time and polylog space, i.e., if $\mathsf{NL} \subseteq \mathsf{SC}$ where SC is Steve's class. In fact, it is not known if an NL-complete language (e.g., directed *s*-*t* connectivity) can be solved simultaneously in polynomial time and strongly sub linear space (i.e., space $n^{1-\alpha}$ for some fixed constant $\alpha > 0$). Thus, studying special problems such as ED, LCS, and LIS, and the relaxed version of approximation is a reasonable first step towards major open problems.

We show that we can indeed achieve our goals. Specifically, for all three problems ED, LCS, and LIS, we give efficient *deterministic* approximation algorithms that can achieve $1+\varepsilon$ or $1-\varepsilon$ approximation, using significantly smaller space than all previous works. In fact, we can even achieve polylog(n) space while maintaining a polynomial running time. By relaxing the space complexity to n^δ for any constant $\delta > 0$, we obtain algorithms whose running time is essentially the same or only slightly more than the standard dynamic programming approach. This is in sharp contrast to the time complexity of ED and LCS, where we only know how to beat the standard dynamic programing significantly by using *randomized* algorithms.

The space efficient approximation algorithms first appeared in [17] and [18]. We present these results and proofs in Chapter 2.

1.1.1 Discussion about Related Work

Classical dynamic programming algorithms can solve both ED and LCS in quadratic time [19]. Currently, the asymptotically fastest algorithm for ED and LCS is given by Grabowsky [20], which runs in $O(\frac{n^2 \log\log n}{\log^2 n})$ time. Recent results in fine grained complexity [2, 3, 21] show that a truly subquadratic time ($O(n^{2-\varepsilon})$ time for some constant $\varepsilon > 0$) algorithm for ED or LCS would refute the Strong Exponential Time Hypothesis (SETH) [4], a conjecture widely believed in the community. [22] further proves that a truly subquadratic time deterministic algorithm that approximate LCS to within a constant factor would imply new lower bound in circuit complexity. For edit distance, many results have been achieved in designing fast approximation algorithms [23–26]. Remarkably, a recent line of study [5–8] shows that ED can be approximated to within a constant factor in near linear time. LCS has also received tremendous attention in recent years [9–11, 22, 27, 28]. Only trivial solutions were known for LCS until very recently: a 2 approximate solution when the alphabet is binary and an $O(\sqrt{n})$ approximate solution for general alphabets in linear time. Both bounds are recently improved by [9] and Rubinstein and Song [11]. Rubinstein et. al. [10] further provide a randomized algorithm in truly subquadratic time with an approximation factor of $O(\lambda^3)$ for LCS , where λ is the ratio of the optimal solution size over the input size.

For LIS, the patience sorting algorithm can solve the problem exactly with $O(n \log n)$ time and $O(n \log n)$ bits of space [29]. It is known that the $O(n \log n)$ time is optimal in the comparison based model [30]. For approximation, Saks and Seshadhri [31] show that there is a randomized algorithm that gives εn additive approximation to LIS in $\mathrm{polylog}\, n$ time. [10] also gives a randomized algorithm in truly sublinear time with an approximation factor of $O(\lambda^3)$ for LIS, where λ is the ratio of the optimal solution size over the input size.

For the space complexity, Savitch's theorem [16] shows that all three problems can be solved in $\mathrm{polylog}\, n$ space with quasi-polynomial time. For LIS, Kiyomi et. al. [15] shows that, for any $\sqrt{n} \le s \le n$, there is a deterministic algorithm that computes LIS using $O(\frac{1}{s} \cdot n^2 \log n)$ time and $O(s \log n)$ bits of space. The authors also prove that such a tradeoff between time and space complexity is optimal in the *sequential access* model. The $O(\sqrt{n}\log n)$ space algorithm is currently the best known polynomial time algorithm for LIS in terms of the space complexity. For approximation, Naumovitz et. al. [32] show that there is a deterministic algorithm that can approximate *distance to monotocity* ($n - \mathsf{LIS}$) to within a $1+\varepsilon$ factor using only $\operatorname{polylog} n$ space, which implies an εn additive approximation of LIS with $\operatorname{polylog} n$ space. For ED and LCS, [33] gives a deterministic algorithm that can compute ED and LCS exactly in $O(n^3)$ time with $O(\frac{n\log^{1.5} n}{2^{\sqrt{\log n}}})$ bits of space. So far, no truly sublinear space ($O(n^{1-\varepsilon})$ space for some constant $\varepsilon > 0$) algorithm that can compute ED or LCS exactly in polynomial time is known.

Approximation algorithms for ED and LCS have also been studied in the massively parallel computation (MPC) model [34, 35]. In the MPC model, algorithms run in a number of rounds. In each round, every machine computes on the data assigned to it and no communication is allowed during a round. Machines can communicate between two rounds.The number of machines and the memory of each machine is relatively smaller then the input size. The goal is to design algorithms with better tradeoffs between the number of machines, the memory of each machine, and the number of rounds. The main result of [34] shows that both ED and LCS can be approximated to within a $1+\varepsilon$ factor with a constant number of rounds while the total running time over all machines is $\tilde{O}(n^2)$. Boroujeni et. al. [35] further improve the number of machines needed for the algorithm of ED.

More discussion about ED, LCS, and LIS in the streaming model is given in Section 1.2.

1.2 Computing String Measures in the Asymmetric Streaming Model

This thesis further studies algorithms and lower bounds for ED and LCS in the asymmetric streaming model. We now give an introduction to the asymmetric streaming model.

In the era of information explosion, ED and LCS are often studied on very large strings. For example, in bioinformatics a human genome can be represented as a string with 3 billion letters (base pairs). Such data provides a huge challenge to the algorithms for ED and LCS, as the standard algorithms for these two problems using dynamic programming need $\Theta(n^2)$ time and $\Theta(n)$ space where n is the length of each string. These bounds quickly become infeasible or too costly as n becomes large, such as in the human genome example. Especially, some less powerful computers may not even have enough memory to store the data, let alone processing it.

One appealing approach to dealing with big data is designing *streaming algorithms*, which are algorithms that process the input as a data stream. Typically, the goal is to compute or approximate the solution by using sublinear space (e.g., n^α for some constant $0 < \alpha < 1$ or even $\mathrm{polylog}(n)$) and a few (ideally one) passes of the data stream. These algorithms have become increasingly popular, and attracted a lot of research activities recently.

Designing streaming algorithms for ED and LCS, however, is not an easy task. For ED, only a couple of positive results are known when we assume the edit distance is much smaller than the input size. We give more discussion in Section 1.2.1. For LCS, strong lower bounds are given in [36, 37], which show that for exact computation, even constant pass randomized algorithms need space $\Omega(n)$; while any constant pass deterministic algorithm achieving a $\frac{2}{\sqrt{n}}$ approximation of LCS also needs space $\Omega(n)$, if the alphabet size is at least n.

In [38], the authors proved a lower bound on the *query complexity* for computing ED in the *asymmetric query* model, where one have random access to one string but only limited number of queries to the other string.

Motivated by the situation that designing streaming algorithms for ED and LCS is hard and inspired by the work of [38], Saks and Seshadhri [39] studied the asymmetric data streaming model. This model is a relaxation of the standard streaming model, where one has streaming access to one string (say x), and random access to the other string (say y). In this model, [39] gives a deterministic one pass algorithm achieving a $1+\varepsilon$ approximation of $n-\mathsf{LCS}$ (which is half of the the edit distance between two strings of length n when insertions and deletions, but not substitutions, are allowed) using space $O(\sqrt{(n \log n)/\varepsilon})$, as well as a randomized one pass algorithm algorithm achieving an εn *additive* approximation of LCS using space $O(k \log^2 n/\varepsilon)$ where k is the maximum number of times any symbol appears in y. Another work by Saha [40] also gives an algorithm in this model that achieves an εn *additive* approximation of ED using space $O(\frac{\sqrt{n}}{\epsilon})$.

The asymmetric streaming model is interesting for several reasons. First, it still inherits the spirit of streaming algorithms, and is particularly suitable for a distributed setting. For example, a local, less powerful computer can use the streaming access to process the string x, while sending queries to a remote, more powerful server which has access to y. Second, because it is a relaxation of the standard streaming model, one can hope to design better algorithms for ED or to beat the strong lower bounds for LCS in this model.

In this thesis, we study both lower bounds and upper bounds for the space complexity of ED and LCS in the asymmetric streaming model. Our lower bounds also extend to *longest increasing subsequence* (LIS) and *longest non-decreasing subsequence* (LNS). We present our algorithms in Chapter 3 and prove several space lower bounds in Chapter 4. These results first appeared in [18] and [41].

1.2.1 Discussion about Related Work

We summarize some of the results about algorithms for ED in the streaming model now. Assuming that the edit distance between the two strings is bounded by some parameter k, [12] gives a randomized one pass algorithm achieving an $O(k)$ approximation of ED, using linear time and $O(\log n)$ space, in a variant of the streaming model where one can scan the two strings simultaneously in a coordinated way. In the same model [12] also give randomized one pass algorithms computing ED exactly, using space $O(k^6)$ and time $O(n+k^6)$. This was later improved to space $O(k)$ and time $O(n+k^2)$ in [12, 13]. Furthermore, [13] give a randomized one pass algorithm computing ED exactly, using space $\tilde{O}(k^8)$ and time $\tilde{O}(k^2 n)$, in the standard streaming model. We note that all of these algorithms are only interesting if k is small, e.g., $k \leq n^\alpha$ where α is some small constant, otherwise the space complexity can be as large as n.

LIS has also been studied in the streaming model. The patience sorting algorithm only needs to read the input sequence from left to right once. Assuming the LIS is k, then patience sorting provides a one-pass streaming algorithm that computes LIS exactly in $O(n \log k)$ time and $O(k \log n)$ bits of space. [36, 37] give a matching lower bound, which holds even for any randomized algorithm with a constant number of passes. Gopalan et. al. [14] present a deterministic one-pass streaming algorithm that can output a $1+\varepsilon$ approximation of LIS with $O(\sqrt{n/\varepsilon}\log n)$ bits of space. Subsequently, [42, 43] prove a lower bound of $\Omega(\frac{1}{R}\sqrt{n/\varepsilon}\log\frac{|\Sigma|}{n})$ for any deterministic streaming algorithm with R passes, where Σ is the alphabet set of the input. The lower bound essentially matches the upper bound. However, their technique does not extend to randomized algorithms [44]. Whether there exists a randomized algorithm with a constant number of passes that can give constant approximation of LIS using $o(\sqrt{n})$ space is still an open problem. Another note worthy result is by Naumovitz et. al. [32]. The paper gives a one-pass deterministic streaming algorithm that can approximate *distance to monotocity* ($n -$ LIS) to within a $1+\varepsilon$ factor using only polylog n space, which implies an εn additive approximation of LIS with polylog n space.

1.3 Locally Decodable Codes for Edit Errors

Error correcting codes are fundamental mathematical objects in both theory and practice, whose study dates back to the pioneering work of Shannon and Hamming in the 1950's. While the study of classical codes focuses on Hamming errors, many exciting variants have emerged ever since. One of the variants studies the *edit* error (or *synchronization* error in some literature), namely *insertions* and *deletions* (*insdels*, for short). Edit errors are strictly more general than Hamming errors and happen frequently in various applications such as text/speech processing, media access, and communication systems. The study of codes for edit errors (*insdel codes*, for short) also has a long history that goes back to the work of Levenstein [45] in the 1960's.

Another line of work studies error correcting codes with the *local decoding* property, which can decode any message symbol by querying only a few codeword symbols.

In this thesis, we focuse on locally decodable codes for edit errors, which we call *Insdel LDCs*. More formally, Locally Decodable Codes (LDCs) are error-correcting codes $C : \Sigma^n \to \Sigma^m$ that allow very fast recovery of individual symbols of a *message* $x \in \Sigma^n$, even when worst-case errors are introduced in the encoded message, called *codeword* $C(x)$.

The important parameters of LDCs are their *rate*, defined as the ratio between the message length n and the codeword length m, measuring the amount of redundancy in the encoding; their *relative minimum distance*, defined as the minimum normalized Hamming distance between any pair of codewords, a parameter relevant to the fraction of correctable errors; and their *locality* or *query complexity*, defined as the number of queries a decoder makes to a received word $y \in \Sigma^m$ in order to decode the symbol at location $i \in [n]$ of the message, namely x_i.

Since they were introduced in [46, 47], LDCs have found many applications in private information retrieval, probabilistically checkable proofs, self-correction, fault-tolerant circuits, hardness amplification, and data structures (e.g., [48–54] and surveys [55, 56]), and the tradeoffs between the achievable parameters of Hamming LDCs has been studied extensively [57–70] (see also surveys by Yekhanin [62] and by Kopparty and Saraf [71]). This sequence of results has brought up exciting progress regarding the necessary and sufficient rate for codes with small query complexity that can withstand a constant fraction of errors.

In this thesis, we present strong lower bounds for Insdel LDCs, which in turn provide a strong separation between Hamming LDCs and Insdel LDCs. We also introduce the notion of LDCs with randomized encoding. We show that under this relaxed definition of LDCs, with the use of randomness in the encoding process, we can achieve constructions with much better query-rate tradeoffs.

1.3.1 Lower Bounds

Despite the extensive research, many important parameter regimes leave wide gaps in our current understanding of LDCs. For example, even for 3-query Hamming LDCs the gap between constructions and lower bounds is superpolynomial [46, 60, 61, 63, 64, 72]. (Note: [65] established an exponential lower bound on the length of 3-query LDCs for some parameter regimes, but it does not rule out the possibility of a 3-query LDC with polynomial length in natural parameter ranges.)

More specifically, for 2-query Hamming LDCs we have matching upper and lower bounds of $m = 2^{\Theta(n)}$, where the upper bound is achieved by the simple Hadamard code while the lower bound is established in [57, 73]. In the constant-query regime where the decoder makes 2^t many queries, for some $t > 1$, the best known constructions of Hamming LDCs are based on matching-vector codes, and give codes that map n

symbols into $m = \exp(\exp((\log n)^{1/t}(\log\log n)^{1-1/t}))$ symbols [61, 63, 64], while the best general lower bound for q-query LDC is $\Omega(n^{\frac{q+1}{q-1}})/\log n$ when $q \geq 3$ [60].

In the polylog(n)-query regime, Reed-Muller codes are examples of $\log^c n$-query Hamming LDCs of block length $n^{1+\frac{1}{c-1}+o(1)}$ for some $c > 0$ (e.g., see [62]). Finally, there exist sub-polynomial (but super logarithmic)-query Hamming LDCs with constant rate [69]. These latter constructions improve upon the previous constant-rate codes in the n^ϵ-query regime achieved by Reed-Muller codes, and upon the more efficient constructions of [74].

Regarding insdel codes, following the work of Levenstein [45], the progress has historically been slow, due to the fact that synchronization errors often result in the loss of index information. Indeed, constructing codes for edit errors is strictly more challenging than for Hamming errors. However the interest in these codes has been rekindled lately, leading to a wave of new results [75–92] (See also the excellent surveys of [93–96]) with almost optimal parameters in various settings, and the variant of "list-decodable" insdel codes, that can withstand a larger fraction of errors while outputting a small list of potential codewords [80, 89, 90]. However, none of these works addresses Insdel LDCs, which we believe are natural objects in the study of insdel codes, since such codes are often used in applications involving large data sets.

Insdel LDCs were first introduced in [97] and further studied in [98–101]. In [97, 98] the authors give Hamming to insdel reductions which transform any Hamming LDC into an Insdel LDC. These reductions decrease the rate by a constant multiplicative factor and increase the locality by a $\log^{c'}(m)$ multiplicative factor for a fixed constant $c' > 1$. Applying the compilers to the above-mentioned constructions of Reed-Muller codes gives $(\log n)^{c+c'}$-query Insdel LDCs of length $m = n^{1+\frac{1}{c-1}+o(1)}$, for any $c > 1$. Also, applying the compilers to the LDCs in [69] yields Insdel LDCs of constant rate and $\exp(\tilde{O}(\sqrt{\log n}))$-query complexity.

Unfortunately, these compilers do not imply constant-query Insdel LDCs, and in fact, even after this work, we do not know if constant-query Insdel LDCs exist in general.

We now formally define the notion of Insdel LDCs.

Definition 1.3.1. *[Insdel Locally Decodable Codes (Insdel LDCs)] Fix an integer q and constants $\delta \in [0,1]$, $\varepsilon \in (0, \frac{1}{2}]$. We say $C\colon \{0,1\}^n \to \Sigma^m$ is a (q,δ,ε)-*locally decodable insdel code *if there exists a probabilistic algorithm* Dec *such that:*

- *For every $x \in \{0,1\}^n$ and $y \in \Sigma^{m'}$ such that $\widetilde{\mathsf{ED}}(C(x), y) \le \delta \cdot 2m$, and for every $i \in [n]$, we have*

$$\Pr\Big[\mathsf{Dec}(y, m', i) = x_i\Big] \ge \frac{1}{2} + \varepsilon,$$

  where the probability is taken over the randomness of Dec*, and $\widetilde{\mathsf{ED}}(C(x), y)$ is the minimum number of insertions/deletions necessary to transform $C(x)$ into y.*

- *In every invocation,* Dec *reads at most q symbols of y. We say that* Dec *is* non-adaptive *if the distribution of queries of* $\mathsf{Dec}(y, m', i)$ *is independent of y.*

We note that $\widetilde{\mathsf{ED}}$ in the definition is usually called the insertion deletion distance. It is slightly different from the edit distance since it does not include substitution as a type of edit operation. Notice that each substitution can be replaced with one insertion and one deletion. For any two strings x and y, we have $\mathsf{ED}(x,y) \le \widetilde{\mathsf{ED}}(x,y) \le 2\,\mathsf{ED}(x,y)$. Also note that in this definition we allow the decoder to have as an input m', the length of the string y. This only makes our lower bounds stronger.

In this thesis, we focus on *binary* Insdel LDCs and give the first non-trivial lower bounds for such codes. In most cases, such as constant-query Insdel LDCs, our bounds are exponential in the message length. We note that prior to our work, the only known lower bounds for Insdel LDCs come from the lower bounds for Hamming LDCs (since Hamming erros can be implemented by edit errors), and thus there is no separation of

Insdel LDC and Hamming LDC. In particular, these bounds don't even preclude the possibility of a 3-query Insdel LDC with $m = \Theta(n^2)$. We also note that we mainly prove lower bounds for Insdel LDCs with non-adaptive decoders. However, by using a reduction suggested in [46] we obtain almost the same lower bounds for Insdel LDCs with adaptive decoders.

Our results provide a strong separation between Insdel LDCs and Hamming LDCs in the following contexts. First, our exponential lower bounds hold for any constant q, while even for $q = 3$ we have constructions of Hamming LDCs with sub-exponential length. Second, for $q = 2$ we rule out the possibility of linear Insdel LDCs, while the Hadamard code is a simple 2-query Hamming LDC. This separation is in sharp contrast to the situation of unique decoding with codes for Hamming errors vs. codes for edit errors, where they have almost the same parameter tradeoffs.

Some further discussions about our lower bounds are provided below.

Implication of Our Lower Bounds to Insdel LCCs Locally Correctable Codes (LCCs) are error-correcting codes $C : \Sigma^n \to \Sigma^m$ that allow very fast recovery of individual symbols of the codeword $C(x) \in \Sigma^m$, even when worst-case errors are introduced. We remark that the lower bounds for Hamming LCCs are asymptotically the same as for LDCs due to a folklore reduction between the two notions (e.g. formalized in [102, 103]). By a similar reduction, our lower bounds imply similar strong lower bounds for Insdel LCCs. Please see [101] for a formal proof..

Private-key LDCs The private-key setting was first studied in [104]. It assumes the encoder and decoder share secret randomness. In contrast to the lower bounds from [57, 73] for LDCs against Hamming error, our lower bounds extend to the private-key setting. We note that one *can* easily obtain private-key Hamming LDCs with $m = \tilde{O}(n)$ and locality 1 by modifying the construction of [104]. Also, applying the compilers from [97] to [104] yields a private-key Insdel LDC with constant rate and locality polylog(n) [99, 100]. Please see [101] for more discussion about the lower bounds for Private-key LDCs.

Motivation of Insdel LDCs in DNA storage DNA storage [105] is a storage medium that harnesses the biology of DNA sequences, to store and transmit not only genetic information, but also any arbitrary digital data, despite the presence of insertion and deletion errors. It has the potential of becoming the storage medium of the future, due to its superior scaling properties, provided new techniques for random data access are developed. Recent progress towards achieving effective and reliable DNA random access technology is motivated by the fact that a *“crucial aspect of data storage systems is the ability to efficiently retrieve specific files or arbitrary subsets of files.”* [1]. This is also precisely the real-world goal formalized by the notion of Insdel LDCs, which motivates a systematic theoretical study of such codes and of their limitations.

The formal statements and proofs of our lower bounds are given in Chapter 5. These lower bounds first appeared in [101].

1.3.2 LDCs with Randomized Encoding

We now introduce the notion of LDCs with randomized encoding. Since the known constructions of LDCs either need a lot of redundancy information, or need to use relatively large number of queries, which is undesirable, closing this gap is an important open problem, which evidently appears to be quite hard.

Note that to ensure the property of local decoding, it is necessary to change the decoding from a deterministic algorithm into a randomized algorithm, since otherwise an adversary can just corrupt all the bits the decoding algorithm queries. Hence, the decoding becomes probabilistic and allows some small probability of failure. With the goal of improving the rate of LDCs in mind, we study a relaxed version of LDCs which is also equipped with randomized encoding. Now the probability of decoding failure is measured over the randomness of both the encoding and decoding.

There are several questions that we need to clarify in this new model. The first question is: *Does the decoding algorithm know and use the randomness of the encoding?* Although we cannot rule out the possibility of constructions where the decoding algorithm can succeed *without* knowing the randomness of the encoding, we only consider the most natural setting, where the decoding algorithm indeed knows and uses the randomness of encoding. After all, without the encoding's randomness the decoding algorithm does not even completely know the encoding function. The second question, as we are considering an adversarial situation, is: *Is the adversary allowed to know the randomness of the encoding?*

This turns out to be a very interesting question. Of course, an adversary who knows the randomness of encoding is much stronger, and thus we need a stronger construction of codes as well. In this thesis, we provide a partial answer to this question, by showing that under some reasonable assumption of the code, a locally decodable code in our model where the adversary knows the randomness of encoding is equivalent (up to constant factors) to a standard locally decodable codes. The assumption is that the code has the following homogenous property:

Property ($\star$) : For any fixing of the encoding's randomness, any fixed error pattern, and any fixed target message bit, the success probability (over the decoding's randomness) of decoding this bit is the same for all possible messages.

We note that this property is satisfied by all known constructions of standard locally decodable codes. Part of the reason is that all known constructions are linear codes, and the decoding only uses non adaptive queries. We now have the following theorem.

Theorem 1.2. *Suppose there is an LDC with randomized encoding, with message length n, codeword length m, that can tolerate δ fraction of errors (either Hamming or edit) and successfully decode any message bit with probability $1 - \varepsilon$ using q queries. Then there also exists a (possibly non explicit) standard LDC, with message length $n/2$, codeword length m, that can tolerate δ fraction of errors and successfully decode any message bit with probability $1 - 2\varepsilon$ using q queries.*

Proof. Fix a specific random string used by the encoding, and a target message bit for decoding (say bit i). Since now the encoding becomes a deterministic function, and there are only finite number of possible error patterns, there is some error pattern that is the *worst* for decoding, i.e., the one that minimizes the success probability of decoding bit i. We say the fixed random string is *good* for bit i if this success probability is at least $1 - 2\varepsilon$. Note that the overall success probability of decoding bit i is at least $1 - \varepsilon$, thus by a Markov argument the probability that a random string is good for bit i is at least $1/2$.

Now again, by an averaging argument, there exists a fixed string that is good for at least $1/2$ fraction of the message bits. Fix this string and the encoding now becomes deterministic. Without loss of generality assume that the string is good for the first half of the message bits. Now fix the rest of the message bits to any arbitrary string (e.g, all 0), we now have a standard LDC with message length $n/2$, codeword length m, that can tolerate δ fraction of errors and successfully decode any message bit with probability $1 - 2\varepsilon$ using q queries. □

The above theorem shows that, in order to get significantly better rate-query tradeoff, we need to either forbid the adversary to know the randomness of encoding, or to construct codes that do not satisfy property ($\star$) (e.g., using adaptive encoding or adaptive decoding). We study the first setting, which is naturally simpler, and we leave codes in the second setting as an interesting open problem. In general, there are two different models where the adversary is not allowed to know the randomness of encoding:

Shared randomness In this model, the encoder and the decoder share a private uniform random string. Thus, the adversary does not know the randomness used by the encoder; but he can add arbitrary errors to the codeword, including looking at the codeword first and then adaptively add errors.

Oblivious channel In this model, the encoder and the decoder do *not* share any randomness. However, the communication channel is oblivious, in the sense that the adversary can add any error pattern *non adaptively*, i.e., without looking at the codeword first.

We can now give our formal definitions of LDCs with randomized encoding.

Definition 1.3.2. *[LDC with a fixed failure probability]*

An $(m, n, \delta, q, \varepsilon)$ LDC with randomized encoding consists of a pair of randomized functions $\{\mathsf{Enc}, \mathsf{Dec}\}$, such that:

- $\mathsf{Enc} : \{0,1\}^n \to \{0,1\}^m$ *is the encoding function. For every message $x \in \{0,1\}^k$, $y = \mathsf{Enc}(x) \in \{0,1\}^n$ is the corresponding codeword.*
- $\mathsf{Dec} : [n] \times \{0,1\}^* \to \{0,1\}$ *is the decoding function. If the adversary adds at most δn errors to the codeword, then for every $i \in [n]$, every $y \in \{0,1\}^*$ which is a corrupted codeword,*

$$\Pr[\mathsf{Dec}(i, y) = x_i] \geq 1 - \varepsilon,$$

where the probability is taken over the randomness of both Enc *and* Dec.

- Dec *makes at most q queries to y.*

Remark 1.3. *Two remarks are in order. First, our definition is quite general in the sense that we don't restrict the type of errors although we focus on edit errors in* *this thesis. Second, as mentioned before, the model can either assume shared private randomness or an oblivious adversarial channel.*

The above definition is for a fixed failure probability ε. However, sometimes it is desirable to achieve a smaller failure probability. For standard locally decodable codes, this can usually be done by repeating the decoding algorithm independently for several times, and then taking a majority vote. This decreases the failure probability at the price of increasing the query complexity. In contrast, in our new model, this approach is not always feasible. For example, in the extreme case one could have a situation where for some randomness used by the encoding, the decoding succeeds with probability 1; while for other randomness used by the encoding, the decoding succeeds with probability 0. In this case repeating the decoding algorithm won't change the failure probability. To rule out this situation, we also define a locally decodable codes with *flexible* failure probability.

Definition 1.3.3. *[LDC with flexible failure probabilities]*

An (m, n, δ) LDC with randomized encoding and query complexity function $q : \mathbb{N} \times [0,1] \to \mathbb{N}$, consists of a pair of randomized algorithms $\{\mathsf{Enc}, \mathsf{Dec}\}$, such that:

- $\mathsf{Enc} : \{0,1\}^n \to \{0,1\}^m$ *is the encoding function. For every message $x \in \{0,1\}^n$, $y = \mathsf{Enc}(x) \in \{0,1\}^m$ is the corresponding codeword.*

- $\mathsf{Dec} : [n] \times \{0,1\}^* \to \{0,1\}$ *is the decoding function. If the adversary adds at most δn errors to the codeword, then for every $i \in [n]$, every $y \in \{0,1\}^*$ which is a corrupted codeword, and every $\varepsilon \in (0,1]$,*

$$\Pr[\mathsf{Dec}(i, y) = x_i] \geq 1 - \varepsilon,$$

  while Dec *makes at most $q = q(m, \varepsilon)$ queries to y. The probability is taken over the randomness of both* Enc *and* Dec.

Again, this definition can apply to both Hamming errors and edit errors, and both the model of shared randomness and the model of an oblivious adversarial channel.

In Chapter 5, we provide several constructions with rate-query tradeoffs significantly better than the standard LDCs. Specifically, we show that for the definition with fixed failure probability, in both the shared randomness model and the oblivious channel model, there are constructions of LDCs with randomized encoding for edit errors that has constant rate and query complexity $q = \mathrm{polylog}(n)\log(1/\varepsilon)$. And we can achieve flexible failure probability with codeword length $m = O(n\log n)$ and query complexity $q = \mathrm{polylog}(n)\log(1/\varepsilon)$ in both models. These results first appeared in [100].

1.3.3 Discussion about Related Work

[104] gives private-key constructions of LDCs with constant rate $m = \Theta(n)$ and locality $\mathrm{polylog}(n)$. [106] extends the construction from [104] to settings where the sender/decoder do not share randomness, but the adversarial channel is resource bounded i.e., there is a safe-function that can be evaluated by the encoder/decoder but not by the channel due to resource constraints (space, computation depth, etc.). By contrast, in the classical setting with no shared randomness and a computationally unbounded channel there are no known constructions with constant rate $m = \Theta(n)$ and locality $\mathrm{polylog}(n)$. [99] applies the [98] compiler to the private-key Hamming LDC of [104] (resp. resource bounded LDCs of [106]) to obtain private-key Insdel LDCs (resp. resource bounded Insdel LDCs) with constant rate and $\mathrm{polylog}(n)$ locality.

Insdel LDCs have also been recently studied in *computationally bounded channels*, introduced in [107]. Such channels can perform a bounded number of adversarial errors, but do not have unlimited computational power as the general Hamming channels. Instead, such channels operate with bounded resources: for example, they might only behave like probabilistic polynomial time machines, or log space machines, or they may only corrupt codewords while being oblivious to the encoder's random coins, or they might have to deal with settings in which the sender and receiver exchange cryptographic keys. As expected, in many such limited-resource settings one can construct codes with strictly better parameters than what can be done generally [108–111]. LDCs in these channels under Hamming error were studied in [104, 106, 112–115].

[99] applies the [98] compiler to the Hamming LDC of [106] to obtain a constant rate Insdel LDCs with polylog(n) locality for resource bounded channels.

Locality in the study of insdel codes was also considered in [81], which constructs explicit synchronization strings that can be locally decoded. Synchronization strings are powerful ingredients that have been used extensively in constructions of insdel codes. In fact, by combining locally decodable synchronization strings with Hamming LDCs, it seems possible to get similar reductions to Insdel LDCs as those in [98, 104].

In our constructions of LDCs with randomized encoding, we protect the message using permutation and random masking. These techniques has been studied in several previous works, e.g. [107, 116].

Chapter 2

Space Efficient Approximation Algorithms

2.1 Introduction

In this chapter, we present space efficient algorithms for approximating edit distance (ED), longest common subsequence (LCS), and longest increasing susbequence (LIS).

2.1.1 Main Results

We first formally define the problems we studied.

Edit Distance The *edit distance* (or *Levenshtein distance*) between two strings $x, y \in \Sigma^*$, denoted by $\mathsf{ED}(x, y)$, is the smallest number of edit operations (insertion, deletion, and substitution) needed to transform one into another. The insertion (deletion) operation adds (removes) a character at some position. The substitution operation replace a character with another character from the alphabet set Σ.

Longest Common Subsequence We say the string $s \in \Sigma^t$ is a *subsequence* of $x \in \Sigma^n$ if there exists indices $1 \leq i_1 < i_2 < \cdots < i_t \leq n$ such that $s = x_{i_1} x_{i_2} \cdots x_{i_t}$. A string s is called a *common subsequence* of strings x and y if s is a subsequence of both x and y. Given two strings x and y, we denote the length of the longest common subsequence (LCS) of x and y by $\mathsf{LCS}(x, y)$.

Longest Increasing Subsequence In the longest increasing subsequence problem, we assume there is a given total order on the alphabet set Σ. We say the string $s \in \Sigma^t$ is an *increasing subsequence* of $x \in \Sigma^n$ if there exists indices $1 \leq i_1 < i_2 < \cdots < i_t \leq n$ such that $s = x_{i_1}x_{i_2}\cdots x_{i_t}$ and $x_{i_1} < x_{i_2} < \cdots < x_{i_t}$. We denote the length of the longest increasing subsequence (LIS) of string x by $\mathsf{LIS}(x)$. In our analysis, for a string x of length n, we assume each element in the string can be stored with space $O(\log n)$. For analysis, we introduce two special symbols ∞ and $-\infty$ with $\infty > i$ and $-\infty < i$ for any character $i \in \Sigma$. In our discussion, we let ∞ and $-\infty$ to be two imaginary characters such that $-\infty < \alpha < \infty$ for all $\alpha \in \Sigma$.

Longest Non-decreasing Subsequence The longest non-decreasing subsequence is a variant of the longest increasing problem. The difference is that in a non-decreasing subsequence $s = x_{i_1}x_{i_2}\cdots x_{i_t}$, we only require $x_{i_1} \leq x_{i_2} \leq \cdots \leq x_{i_t}$.

Notations We use the following conventional notations. Let $x \in \Sigma^n$ be a string of length n over alphabet Σ. By $|x|$, we mean the length of x. We denote the i-th character of x by x_i and the substring from the i-th character to the j-th character by $x[i:j]$. We denote the concatenation of two strings x and y by $x \circ y$. By $[n]$, we mean the set of positive integers no larger than n.

We now formally state our results. For edit distance, we have the following theorem.

Theorem 2.1. *Given any strings x and y each of length n, there are deterministic algorithms that approximate $\mathsf{ED}(x, y)$ with the following parameters:*

1. *For any constants $\delta \in (0, \frac{1}{2})$ and $\varepsilon \in (0, 1)$, an algorithm that outputs a $1+\varepsilon$ approximation of $\mathsf{ED}(x, y)$ using $\tilde{O}_{\varepsilon,\delta}(n^\delta)$ bits of space and $\tilde{O}_{\varepsilon,\delta}(n^2)$ time.*
2. *An algorithm that outputs a $1 + O(\frac{1}{\log\log n})$ approximation of $\mathsf{ED}(x, y)$ using $O(\frac{\log^4 n}{\log\log n})$ bits of space and $O(n^{7+o(1)})$ time.*

Note that our second algorithm for ED uses roughly the same running time as the standard dynamic programming, but much smaller space. Indeed, we can use space

n^δ for any constant $\delta > 0$. This also significantly improves the previous result of [39], which needs to use space $\Omega(\sqrt{n}\log n)$ and only provides a $2+\varepsilon$ approximation for standard ED. With a larger (but still polynomial) running time, we can achieve space complexity $O(\frac{\log^4 n}{\log\log n})$.

Theorem 2.2. *Given any strings x and y each of length n, there are deterministic algorithms that approximate $\mathsf{LCS}(x,y)$ with the following parameters:*

1. *For any constants $\delta \in (0, \frac{1}{2})$ and $\varepsilon \in (0,1)$,an algorithm that outputs a $1-\varepsilon$ approximation of $\mathsf{LCS}(x,y)$ using $\tilde{O}_{\varepsilon,\delta}(n^\delta)$ bits of space and $\tilde{O}_{\varepsilon,\delta}(n^{3-\delta})$ time. Furthermore the algorithm can output such a sequence using $\tilde{O}_{\varepsilon,\delta}(n^\delta)$ bits of space and $\tilde{O}_{\varepsilon,\delta}(n^3)$ time.*

2. *An algorithm that outputs a sequence which is a $1-O(\frac{1}{\log\log n})$ approximation of $\mathsf{LCS}(x,y)$, using $O(\frac{\log^4 n}{\log\log n})$ bits of space and $O(n^{6+o(1)})$ time.*

To the best of our knowledge, Theorem 2.2 is the first $1-\varepsilon$ approximation of LCS using truly sub-linear space, and in fact we can achieve space n^δ for any constant $\delta > 0$ with only a slightly larger running time than the standard dynamic programming approach. We can achieve space $O(\frac{\log^4 n}{\log\log n})$ with an even larger (but still polynomial) running time.

For LIS, we also give efficient deterministic approximation algorithms that can achieve $1-\varepsilon$ approximation, with better space complexity than that of [14] and [15]. In particular, we can achieve space n^δ for any constant $\delta > 0$ and even space $O(\frac{\log^4 n}{\log\log n})$. We have the following theorem.

Theorem 2.3. *Given any string x of length n, there are deterministic algorithms that approximate $\mathsf{LIS}(x)$ with the following parameters:*

1. *For any constants $\delta \in (0, \frac{1}{2})$ and $\varepsilon \in (0,1)$, an algorithm that computes a $1-\varepsilon$ approximation of $\mathsf{LIS}(x)$ using $\tilde{O}_{\varepsilon,\delta}(n^\delta)$ bits of space and $\tilde{O}_{\varepsilon,\delta}(n^{2-2\delta})$ time.*

Furthermore the algorithm can output such a sequence using $\tilde{O}_{\varepsilon,\delta}(n^{\delta})$ bits of space and $\tilde{O}_{\varepsilon,\delta}(n^{2-\delta})$ time.

2. *An algorithm that outputs a sequence which is a $1 - O(\frac{1}{\log\log n})$ approximation of $\mathsf{LIS}(x)$ using $O(\frac{\log^4 n}{\log\log n})$ bits of space and $O(n^{5+o(1)})$ time.*

Note that our theorems show that achieving a $1 \pm O(\frac{1}{\log\log n})$ approximation of $\mathsf{ED}, \mathsf{LCS}, \mathsf{LIS}$ can be done simultaneously in polynomial time and polylog space. Thus these approximation problems are in the class SC.

Remark 2.4. *Our algorithms for LIS also work for the problem of longest non-decreasing subsequence. This is due to the following reduction. Given the original sequence $x \in \Sigma^n$, we change it to a new sequence y where $y_i = (x_i, i) \in \Sigma \times [n]$. We define a new total order on the set $\Sigma \times [n]$ such that $(x_i, i) < (x_j, j)$ if $x_i < x_j$, or, $x_i = x_j$ and $i < j$. Then it is easy to see $\mathsf{LIS}(y)$ is equal to the length of the longest non-decreasing subsequence in x. This reduction is also a logspace reduction.*

2.1.2 Overview of Techniques

The starting point of all our space efficient approximation algorithms is the well known Savitch's theorem [16], which roughly shows that any non-deterministic algorithm running in space $s \geq \log n$ can be turned into a deterministic algorithm running in space $O(s^2)$ by using recursion. Since all three problems of ED, LCS, and LIS can be computed exactly in non-deterministic logspace, this trivially gives deterministic algorithms that compute all of them exactly in space $O(\log^2 n)$. However in the naive way, the running time of these algorithms become quasi-polynomial.

To reduce the running time, we turn to approximation. Here we use two different sets of ideas. The first set of ideas applies to ED. Note that the reason that the above algorithm for computing ED runs in quasi-polynomial time, is that in each recursion we are computing the ED between *all* possible substrings of the two input strings. To avoid this, we use an idea from [34], which shows that to achieve a good approximation, we only need to compute the ED between some carefully chosen substrings of the two input strings. Using this idea in each level of recursion gives us the space efficient approximation algorithms for ED.

The second set of ideas applies to LCS and LIS. Here, we first give a small space reduction from LCS to LIS, and then we can focus on approximating LIS. Again, the reason that the naive $O(\log^2 n)$ space algorithm for LIS runs in quasi-polynomial time, is that in each recursion we are looking at *all* possible cases of breaking the input string into two substrings, computing the LIS in the two substrings which ends and starts at the break point, and taking the maximum of the sums. To get an approximation, we use the *patience sorting* algorithm for computing LIS exactly [29], and the modification in [14] which gives an approximation of LIS using smaller space by equivalently looking at only some carefully chosen cases of breaking the input string into two substrings. The rough idea is then to use the algorithm in [14] recursively, but making this work requires significant modification of the algorithm in [14], both to make the recursion work and to make it work under the reduction from LCS to LIS.

We now give more details below.

2.1.2.1 Edit Distance

As discussed before, our approximation algorithm for ED is based on recursion. In each level of recursion, we use an idea from [34] to approximate the edit distance between certain pairs of substrings. We start by giving a brief description of the algorithm in [34].

Let x and y be two input strings each of length n. Assume we want to get a $(1+\varepsilon)$-approximation of $\mathsf{ED}(x, y)$ where ε is any constant in $(0, 1)$. Let $\delta \in (0, 1)$ be a constant which we choose later, and $\varepsilon' = \varepsilon/10$. The algorithm guesses a value $\Delta \leq n$ which is supposed to be a $(1+\varepsilon')$-approximation of $\mathsf{ED}(x, y)$. If this is true then the algorithm will output a $(1+\varepsilon)$-approximation of $\mathsf{ED}(x,y)$. To get rid of guessing, we can try every $\Delta \le n$ such that $\Delta = \lceil (1+\varepsilon')^i \rceil$ for some integer i and take the minimum. This does not affect the space required, and only increases the time complexity by a logarithmic factor.

The idea is to first divide x into $N = n^\delta$ blocks each of length $n^{1-\delta}$. For simplicity, we fix an optimal alignment between x and y such that $x_{[l_i,r_i]}$ is matched to the substring $y_{[\alpha_i,\beta_i]}$, and the intervals $\{[\alpha_i,\beta_i]\}$ are disjoint and span the entire length of y. We say that an interval $[\alpha',\beta']$ is an (ε,Δ)-*approximately optimal candidate* of the block $x^i = x_{[l_i,r_i]}$ if the following two conditions hold:

$$\alpha_i \le \alpha' \le \alpha_i + \varepsilon\frac{\Delta}{N}$$

$$\beta_i - \varepsilon\frac{\Delta}{N} - \varepsilon\,\mathsf{ED}(x_{[l_i,r_i]}, y_{[\alpha_i,\beta_i]}) \le \beta' \le \beta_i$$

[34] showed that, for each block of x that is not matched to a too large or too small interval in y, there is a way to choose $O_\varepsilon(n^\delta \log n)$ intervals such that one of them is an (ε',Δ)-*approximately optimal candidate*. Then we can compute the edit distance between each block and all of its corresponding candidate intervals, which gives $O_\varepsilon(n^{2\delta}\log n)$ values. After this, we can use dynamic programming to find a $(1+\varepsilon)$-approximation of the edit distance if Δ is a $(1+\varepsilon')$-approximation of $\mathsf{ED}(x,y)$.

Computing the edit distance of each block in x with one of its candidate intervals in y takes $O_\varepsilon(n^{1-\delta}\log n)$ bits of space (we assume each symbol can be stored with space $O(\log n)$). We can run this algorithm sequentially and reuse the space for each computation. Storing the edit distance of each pair takes $\tilde{O}_\varepsilon(n^{2\delta})$ space. Thus, if we take $\delta = 1/3$, the above algorithm uses a total of $\tilde{O}_\varepsilon(n^{2/3})$ bits of space.

We now run the above algorithm recursively to further reduce the space required. Let δ be a small constant in $(0,1)$. Our algorithm takes four inputs: two strings x and y each of length n, $N = n^\delta$, and $\varepsilon \in (0,1)$. The goal is to output a $(1+\varepsilon)$ approximation of $\mathsf{ED}(x,y)$ with $\tilde{O}_{\varepsilon,\delta}(N^2)$ space. Similarly, we first divide x into n^δ

blocks. We try every Δ that is equal to $\lceil (1+\varepsilon')^i \rceil$ for some integer i, and for each Δ there is a set of candidate intervals depending on Δ. Then, for each block of x and each of its $O_{\varepsilon}(n^{\delta} \log n)$ candidate intervals, instead of computing the edit distance exactly, we recursively call our space efficient approximation algorithm with this pair as input, while keeping N unchanged and decreasing ε by a factor of 2. We argue that if the recursive call outputs a $(1+\varepsilon/2)$-approximation of the actual edit distance, the output of the dynamic programming increases by at most a $(1+\varepsilon/2)$ factor. Thus if Δ is a $(1+\varepsilon')$-approximation of $\mathsf{ED}(x, y)$, the output of the dynamic programming is guaranteed to be a $(1+\varepsilon)$-approximation. The recursion stops whenever the input string has length at most N. In this case, we compute the edit distance exactly with $O(n^{\delta} \log n)$ space.

Notice that at each level of the recursion, the first input string is divided into N blocks if it has length larger than N. Thus the length of first input string at the i-th level of recursion is at most $n^{1-\delta i}$. Hence, the depth of recursion is bounded by a constant $d = \frac{1-\delta}{\delta}$. We denote the ε at the i-th level by ε_i, thus we have $\varepsilon_i = \frac{\varepsilon_1}{2^{i-1}}$. Similarly we set $\varepsilon'_i = \varepsilon_i/10$. We show that the output of the i-th level of recursion is a $1+\varepsilon_i$ approximation of the edit distance by induction on i from d to 1. Thus, the output in the first level is guaranteed to be a $(1+\varepsilon)$-approximation. At the i-th level, we either invoke one more level of recursion and maintains $\tilde{O}_{\varepsilon_i,\delta}(N^2)$ values, or do an exact computation of edit distance which takes $O(n^{\delta} \log n)$ space. Hence, the space used at each level is bounded by $\tilde{O}_{\varepsilon_i,\delta}(N^2)$. There are at most $d = \frac{1-\delta}{\delta} + 1$ levels. The aggregated space used by our recursive algorithm is still $\tilde{O}_{\varepsilon,\delta}(N^2)$. For time complexity, we can bound the number of times the algorithm enters the i-th level of recursion by $\tilde{O}_{\varepsilon,\delta}(n^{2\delta(i-1)})$. At the d-th level, an exact computation of edit distance takes $O_{\varepsilon,\delta}(n^{2\delta})$ time. For $i < d$, the computation at the i-th level uses a dynamic programming that taks $O_{\varepsilon,\delta}(n^{3\delta})$ time. Thus, the total time is bounded by $\tilde{O}_{\varepsilon,\delta}(n^2)$.

2.1.2.2 Longest Increasing Subsequence

We now consider the problem of approximating the LIS of a string $x \in \Sigma^n$ over the alphabet Σ which has a total order. We assume each symbol in Σ can be stored with $O(\log n)$ bits of space. For our discussion, we let ∞ and $-\infty$ be two special symbols such that for any symbol $\sigma \in \Sigma$, $-\infty < \sigma < \infty$. We denote the length of the longest increasing subsequence of x by $\mathsf{LIS}(x)$.

Again our algorithm is a recursive one, and in each recursion we use an approach similar to the deterministic streaming algorithm from [14] that gives a $1-\varepsilon$ approximation of $\mathsf{LIS}(x)$ with $O(\sqrt{n/\varepsilon}\log n)$ space. Before describing their approach, we first give a brief introduction to a classic algorithm for LIS, known as $\mathsf{PatienceSorting}$. The algorithm initializes a list P with n elements such that $P[i] = \infty$ for all $i \in [n]$, and then scans the input sequence x from left to right. When reading a new symbol x_i, we find the smallest index l such that $P[l] \geq x_i$ and set $P[l] = x_i$. After processing the string x, for each i such that $P[i] < \infty$, we know $\sigma = P[i]$ is the smallest possible character such there is an increasing subsequence in x of length i ending with σ. We give the pseudocode in algorithm 1 and refer readers to [29] for more details about this algorithm.

Algorithm 1: PatienceSorting

```
Input: A string $x \in \Sigma^n$
1 initialize a list $P$ with $n$ elements such that $P[i] = \infty$ for all $i \in [n]$
2 for $i = 1$ to $n$ do
3     let $l$ be the smallest index such that $P[l] \geq x_i$
4     $P[l] \leftarrow x_i$
5 end
6 let $l$ be the largest index such that $P[l] < \infty$
7 return $l$
```

We have the following result.

Lemma 2.1.1. *Given a string x of length n, PatienceSorting computes LIS(x) in $O(n \log n)$ time with $O(l \log n)$ bits of space where $l =$ LIS(x).*

In the streaming algorithm from [14], we maintain a set S and a list Q, such that, $Q[i]$ is stored only for $i \in S$ and $S \subseteq [n]$ is a set of size $O(\sqrt{n/\varepsilon})$. We can use S and Q as an approximation to the list P in $\mathsf{PatienceSorting}$ in the sense that for each $s \in S$, there is an increasing subsequence in x of length s ending with $Q[s]$. More specifically, we can generate a list P' from S and Q such that $P'[i] = Q[j]$ for the smallest $j \geq i$ that lies in S. For i larger than the maximum element in S, we set $P'[i] = \infty$. Each time we read a new element from the data stream, we update Q and S accordingly. The update is equivalent to doing $\mathsf{PatienceSorting}$ on the list P'. When S gets larger than $2\sqrt{n/\varepsilon}$, we do a cleanup to S by only keeping $\sqrt{n/\varepsilon}$ evenly picked values from 1 to $\max S$ and storing $Q[s]$ for $s \in S$. Each time we do a "cleanup", we lose at most $\sqrt{\varepsilon/n}\,\mathsf{LIS}(x)$ in the length of the longest increasing subsequence detected. Since we only do $\sqrt{\varepsilon n}$ cleanups, we are guaranteed to detect an increasing subsequence of length at least $(1-\varepsilon)\,\mathsf{LIS}(x)$.

We now modify the above algorithm into another form. This time we first divide the input string x into many small blocks. Meanwhile, we also maintain a set S and a list Q. We now process x from left to right, and update S and Q each time we have processed one block of x. If the number of blocks in x is small, we can get the same approximation as in [14] with S and Q having smaller size. For example, we can divide x into $O_\varepsilon(n^{1/3})$ blocks each of size $O_\varepsilon(n^{2/3})$, and we update S and Q once after processing each block. If we do exact computation within each block, we only need to maintain the set S and the list Q of size $O(\sqrt{\varepsilon}n^{1/3})$. We can still get a $(1-\varepsilon)$ approximation, because we do $O(\sqrt{\varepsilon}n^{1/3})$ cleanups and for each cleanup, we lose about $O(\sqrt{\varepsilon}n^{-1/3})\,\mathsf{LIS}(x)$ in the length of the longest increasing subsequence detected.

This almost already gives us an $O_\varepsilon(n^{1/3})$ space algorithm, except the exact computation within each block needs $\Omega_\varepsilon(n^{2/3})$ space. A natural idea to reduce the space complexity is to replace the exact computation with an approximation. When each block x^i has size $O_\varepsilon(n^{2/3})$, running the approximation algorithm from [14] takes

$O_\varepsilon(n^{1/3})$ space and thus we can hope to reduce the total space required to $O_\varepsilon(n^{1/3})$. However, a problem with this is that by running the approximation algorithm on each block x^i, we only get an approximation of $\mathsf{LIS}(x^i)$. This alone does not give us enough information on how to update S and Q. Also, for a longest increasing subsequence of x, say τ, the subsequence of τ that lies in the block x^i may be much shorter than $\mathsf{LIS}(x^i)$. This subsequence of τ may be ignored if we run the approximation algorithm instead of using exact computation.

We now give some intuition of our approach to fix these issues. Let us consider a longest increasing subsequence τ of x such that τ can be divided into many parts, where the i-th part τ^i lies in x^i. We denote the length of τ^i by d_i. Let the first symbol of τ^i be α_i and the last symbol be β_i if τ^i is not empty. When we process the block x^i, we want to make sure that our algorithm can detect an increasing subsequence of length at least $(1-\varepsilon)d_i$ in x^i, where the first symbol is at least α_i and the last symbol is at most β_i. We can achieve this by running a bounded version of the approximation algorithm which only considers increasing subsequences no longer than d_i. Since we do not know α_i or d_i in advance, we can guess α_i by trying every symbol in $Q[s]$ since one of them is close enough to α_i. For d_i, we can try $O(\log_\varepsilon n)$ different values of l such that one of them is close enough to d_i. In this way, we are guaranteed to detect a good approximation of τ_i.

Based on the above intuition, our approach is to recursively build a sequence of algorithms called $\mathsf{ApproxLIS}^d$ for each integer $d \geq 2$ such that $\mathsf{ApproxLIS}^d$ uses only $O_{\varepsilon,d}(n^{\frac{1}{d}} \log n)$ space. The first algorithm $\mathsf{ApproxLIS}^2$ is exactly the same as the algorithm from [14]. Then, we build $\mathsf{ApproxLIS}^{d+1}$ with $\mathsf{ApproxLIS}^d$ as an ingredient.

For each $\mathsf{ApproxLIS}^d$, we introduce a slightly modified version called $\mathsf{ApproxLISBound}^d$. $\mathsf{ApproxLISBound}^d$ takes an additional input l, which is an integer at most n. We want to guarantee that if the string x has an increasing subsequence of length l ending with $\alpha \in \Sigma$, then $\mathsf{ApproxLISBound}^d(x, \varepsilon, l)$ can detect an increasing subsequence of length at least $(1-\varepsilon)l$ ending with some symbol no larger than α (recall that Σ has a total order). The idea is to run $\mathsf{ApproxLIS}^d$ but only consider increasing subsequence of length at most l. $\mathsf{ApproxLISBound}^d$ has the same space and time complexity as $\mathsf{ApproxLIS}^d$.

Assume we are given $\mathsf{ApproxLIS}^d$ and $\mathsf{ApproxLISBound}^d$ such that $\mathsf{ApproxLIS}^d(x, \varepsilon)$ outputs a $(1-\varepsilon)$ approximation of $\mathsf{LIS}(x)$ with $O_{\varepsilon,d}(n^{\frac{1}{d}} \log n)$ space. We now describe how $\mathsf{ApproxLIS}^{d+1}$ works. Similar to the streaming algorithm in [14], we maintain a set S and Q of size $O_\varepsilon(n^{\frac{1}{d+1}})$ as an approximation of the list P when running $\mathsf{PatienceSorting}$. We will show that it is enough to use $O_\varepsilon(n^{\frac{1}{d+1}})$ bits for S and Q, because we only update them $O(\sqrt{\varepsilon} n^{\frac{1}{d+1}})$ times and we lose about $O(\sqrt{\varepsilon} n^{-\frac{1}{d+1}})\,\mathsf{LIS}(x)$ after each update. To achieve this, we first divide x into $N = O(\sqrt{\varepsilon} n^{\frac{1}{d+1}})$ blocks each of size $O(n^{\frac{d}{d+1}}/\sqrt{\varepsilon})$. We denote the i-th block by x^i. Initially, S contains only one element 0 and $Q[0] = -\infty$. We update S and Q after processing each block of x as follows.

For simplicity, we denote S and Q after processing the t-th block by S_t and Q_t. To see how S and Q are updated, we take the t-th update as an example. Given S_{t-1} and Q_{t-1}, we first determine the length of LIS in $x^1 \circ \cdots \circ x^t$ that can be detected based on S_{t-1} and Q_{t-1}. We denote this length by k_t. Notice that, for each $s \in S_{t-1}$, we know there is an increasing subsequence in $x^1 \circ \cdots \circ x^{t-1}$ of length s ending with $Q_{t-1}[s] \in \Sigma$. This gives us $|S_{t-1}|$ increasing subsequences. The idea is to find the best extension of these increasing subsequences in the block x^t and see which one gives us the longest increasing subsequence of $x^1 \circ \cdots \circ x^t$. Since each block is of size $O_\varepsilon(n^{\frac{d}{d+1}})$, we cannot afford to do exact computation, thus we use $\mathsf{ApproxLIS}^d$ instead. For each $s \in S_{t-1}$, we run $\mathsf{ApproxLIS}^d(z^s, \varepsilon/3)$ where z^s is the subsequence of x^t with only symbols larger than $Q_{t-1}[s]$. Finally, we let $k_t = \max_{s \in S_{t-1}}(s + \mathsf{ApproxLIS}^d(z^s, \varepsilon/3))$. Given k_t, we then set S_t to be the $n^{\frac{1}{d+1}}/\sqrt{\varepsilon}$ evenly picked integers from 0 to k_t.

The next step is to compute Q_t. We first set $Q_t[s] = \infty$ for all $s \in S_t$ except $s = 0$, and we set $Q_t[0] = -\infty$. Then, for each $s \in S_{t-1}$ and $l = 1, 1+\varepsilon/3, (1+\varepsilon/3)^2, \ldots, k_t - s$, we run $\mathsf{ApproxLISBound}^d(z^s, \varepsilon/3, l)$. For each $s' \in S_t$ such that $s \leq s' \leq s + l$, we update $Q_t[s']$ if $\mathsf{ApproxLISBound}^d(z^s, \varepsilon/3, l)$ detects an increasing subsequence of length $s' - s$ ending with a symbol smaller than the old $Q_t[s']$. The intuition is that, with the bound l, we may be able to find a smaller symbol in Σ such that there is an increasing subsequence of length l ending with it. This information can be easily ignored if l is a lot smaller than the actual length of LIS in x^t. To see why this is important, let τ be a longest increasing subsequence of x, and let τ^t be the part of τ that lies in the block x^t. The length of τ^t may be much smaller than the length of LIS in x^t. When the bound l is close to $|\tau^t|$, we will be able to detect a good approximation of τ^t by running $\mathsf{ApproxLISBound}^d(z^s, \varepsilon/3, l)$ on z^s for each $s \in S_{t-1}$. Since we do not know the length of τ^t, we will guess it by trying $O_\varepsilon(\log n)$ values of l and always record the optimal $Q_t[s]$ for $s \in S_t$.

Continue doing this, we get S_N and Q_N. $\mathsf{ApproxLIS}^{d+1}$ outputs the largest element in S_N.

To see the correctness of our algorithm, let us consider a longest increasing subsequence τ of x. τ can be divided into N parts such that τ^i lies in x^i although some part may be empty. For our analysis, let P'_t be the list generated by S_t and Q_t in the following way: for every i let $P'_t[i] = Q_t[j]$ for the smallest $j \geq i$ that lies in S_t. If no such j exists, set $P'_t[i] = \infty$. Correspondingly, P_t is the list P after running $\mathsf{PatienceSorting}$ with input $x^1 \circ x^2 \circ \cdots \circ x^t$.

Let $h_t = \sum_{j=1}^{t} |\tau^j|$ and $k_t = \max S_t$, our main observation is the following inequality:

$$P'_t[(1 - \frac{2\varepsilon}{3})h_t - 2t\varepsilon' n^{-\frac{1}{d+1}} k_t] \leq P_t[h_t] \tag{2.1}$$

Note that when $t = N$, $h_N = \mathsf{LIS}(x)$. We have $P_t[h_t] < \infty$ by the correctness of $\mathsf{PatienceSorting}$. If inequality 2.1 holds, there is an element in S_N larger than $(1 - \varepsilon)\,\mathsf{LIS}(x)$ which directly gives the correctness of $\mathsf{ApproxLIS}^{d+1}$.

We prove inequality 2.1 by induction on t. The intuition is that, at the t-th update, by the fact that inequality 2.1 holds for $t-1$, we know that there must exist an $s \in S_{t-1}$ that is close to h_{t-1} and $Q_{t-1}[s] \leq P_{t-1}[h_{t-1}] = \beta_{t-1} < \alpha_t$. By trying $l = 1, 1+\varepsilon/3, (1+\varepsilon/3)^2, \dots, k_t - s$, one l is close enough to $|\tau^t|$. Thus we are guaranteed to detect a good approximation of τ_t in x^t and the inequality also holds for t.

For the space complexity, running $\mathsf{ApproxLIS}^d$ and $\mathsf{ApproxLISBound}^d$ on each block of size $O_\varepsilon(n^{\frac{d}{d+1}})$ uses $O_{\varepsilon,d}(n^{\frac{1}{d+1}})$ space, and storing S, Q uses an additional $O_\varepsilon(n^{\frac{1}{d+1}})$ space. Thus the total space used by $\mathsf{ApproxLIS}^{d+1}$ is still $O_{\varepsilon,d}(n^{\frac{1}{d+1}})$.

Our algorithm for approximating the length of LIS can be modified to output an increasing subsequence. Again, the idea is to build a sequence of algorithms called $\mathsf{LISSequence}^i$ for each integer $i \geq 1$ such that $\mathsf{LISSequence}^i(x, \varepsilon)$ can output an increasing subsequence of x with length at least $(1-\varepsilon)\,\mathsf{LIS}(x)$, using $O_{\varepsilon,i}(n^{\frac{1}{i}} \log n)$ space. For the first algorithm $\mathsf{LISSequence}^1$, we can output the LIS exactly with $O(n \log n)$ space, see [29] for example. Now, assume we are given algorithm $\mathsf{LISSequence}^d$, we show how $\mathsf{LISSequence}^{d+1}$ works. Let ρ be the longest increasing subsequence detected by $\mathsf{ApproxLIS}^{d+1}(x, \varepsilon/2)$, thus ρ has length $(1-\varepsilon/2)\,\mathsf{LIS}(x)$. We divide ρ into N parts such that the i-th part ρ^i lies in x^i, thus ρ^i has length at most $|x^i| = n^{\frac{d}{d+1}}$. If we know the first and last symbol of ρ^i, we can output an increasing subsequence of length at least $(1-\varepsilon/2)|\rho^i|$ by running $\mathsf{LISSequence}^d(x^i, \varepsilon/2)$ while ignoring all symbols in x^i that is smaller than the first symbol of ρ^i or larger than the last symbol of ρ^i. This can be done with $O_{\varepsilon,d}(n^{\frac{1}{d}} \log n)$ space. To determine the range of ρ, that is, the first and the last symbol of each part ρ^i, we compute a list B of $N+1$ symbols $B[0], \cdots, B[N]$. We first set $B[N]$ to be $Q_N[s_N]$ where s_N is the largest element in S_N. This is because s_N is the length of ρ and $B[N]$ is the last symbol of ρ. Then, we compute the list B from right to left. Once we know $B[i] = Q_i[s_i]$ for some $s_i \in S_i$, we compute S_{i-1} and Q_{i-1} by running $\mathsf{ApproxLIS}^{d+1}(x, \varepsilon/2)$ again. Then, for each $s \in S_{i-1}$ and $s \leq s_i$, if we can find an increasing subsequence in x^{i-1} of length $s_i - s$ with the first symbol larger than $Q_{i-1}[s]$ and the last symbol at most $B[i]$, we set $s_{i-1} = s$, $B[i-1] = Q_{i-1}[s_{i-1}]$ and continue to compute $B[i-2]$. After we finish computing B, we can use $B[i-1]$ and $B[i]$ as the range of ρ^i. Computing B needs to run $\mathsf{ApproxLIS}^{d+1}$ for N times sequentially and B is of size $N = O_\varepsilon(n^{\frac{1}{d}} \log n)$, thus the space used is bounded by $O_{\varepsilon,d}(n^{\frac{1}{d}} \log n)$.

2.1.2.3 Longest Common Subsequence

For longest common subsequence, our algorithm is based on a reduction from LCS to LIS. We assume the inputs are two strings $x \in \Sigma^n$ and $y \in \Sigma^m$. Our goal is to output a $(1-\varepsilon)$-approximation of the LCS of x and y.

We first introduce the following reduction from LCS to LIS. Given the strings x and y, for each $i \in [n]$ let $b^i \in [m]^*$ be the sequence consisting of all distinct indices j in $[m]$ such that $x_i = y_j$, arranged in descending order. Note that b^i may be empty. Let z be the sequence such that $z = b^1 \circ b^2 \circ \cdots \circ b^n$, which has length $O(mn)$ since each sequence b^i is of length at most m. We claim that $\mathsf{LIS}(z) = \mathsf{LCS}(x, y)$. This is because for every increasing subsequence of z, say $t = t_1 t_2 \cdots t_d$, the corresponding subsequence $y_{t_1} y_{t_2} \cdots y_{t_d}$ of y also appears in x. Conversely, for every common subsequence of x and y, we can find an increasing subsequence in z with the same length. We call this procedure ReduceLCStoLIS. Note that in our algorithms, z need not be stored, since we can compute each element in z as necessary in logspace by querying x and y. Thus our reduction is a logspace reduction.

Once we reduce the LCS problem to an LIS problem, we can use similar techniques as we use for LIS. We build a sequence of algorithms called $\mathsf{ApproxLCS}^i$ for each integer $i \geq 1$ such that $\mathsf{ApproxLCS}^i(x, y, \varepsilon)$ computes a $(1-\varepsilon)$-approximation of $\mathsf{LCS}(x, y)$ with $O_{\varepsilon,i}(n^{\frac{1}{i}} \log n)$ space. For the first algorithm $\mathsf{ApproxLCS}^1$, we run $\mathsf{PatienceSorting}$ on z to compute $\mathsf{LIS}(z)$ exactly. Since $\mathsf{LIS}(z) = \mathsf{LCS}(x, y) \leq n$, it can be done with

$O(n \log n)$ space.

For $\mathsf{ApproxLCS}^{d+1}$, the goal is to compute an approximation of $\mathsf{LIS}(z)$. We first divide x evenly into $N = O_\varepsilon(n^{\frac{1}{d+1}})$ blocks $x^1, x^2, \ldots, x^N$. Correspondingly the string $z = \text{ReduceLCStoLIS}(x, y)$ can be divided into N blocks with $z^i = \text{ReduceLCStoLIS}(x^i, y)$. We know $\mathsf{LIS}(z^i)$ is at most $O_\varepsilon(n^{\frac{d}{d+1}})$ since the length of x^i is $O_\varepsilon(n^{\frac{d}{d+1}})$. Then we can compute an approximation of $\mathsf{LIS}(z^i)$ with $\mathsf{ApproxLCS}^d$, which takes only $O_{\varepsilon,d}(n^{\frac{1}{d+1}} \log n)$ space. We can now build $\mathsf{ApproxLCS}^{d+1}$ based on $\mathsf{ApproxLCS}^d$ with the same approach as in the construction of $\mathsf{ApproxLIS}^{d+1}$. Since we divide z into N blocks with $z^i = \text{ReduceLCStoLIS}(x^i, y)$, this approach also gives us a slight improvement on running time over the naive approach of running our LIS algorithm after the reduction. Similar to our algorithm for LIS, we can modify our algorithms to output the common subsequence but not only the length.

2.2 Space Efficient Approximation for ED

In this section, we give a formal description of our space efficient algorithm for approximating edit distance. We start with some definitions and tools from [34].

Let $x \in \Sigma^n$ and $y \in \Sigma^m$ be two strings over alphabet Σ. We assume each symbol of Σ can be stored with $O(\log n)$ bits. Our goal is to output a $1 + \varepsilon$ approximation for $\mathsf{ED}(x, y)$. Here, ε is a parameter that can be subconstant. In our algorithm, we only consider the case when $n = \Theta(m)$ since otherwise output a good approximation of $\mathsf{ED}(x, y)$ would be easy.

We assume an integer Δ is given to us which is supposed to be a $1+\varepsilon$ approximation of the actual edit distance. Such an assumption only increases the total amount of computation by a $O(\log_{1+\varepsilon}(n))$ factor. This is because we can try all $\Delta = (1 + \varepsilon)^i$ with $i \in [\lceil \log_{1+\varepsilon}(n) \rceil]$ and make sure one of Δ is a $1 + \varepsilon$ approximation of $\mathsf{ED}(x, y)$.

Given such a Δ, we first divide string x into b blocks (b is a parameter we will pick later). We denote the i-th block by x^i. Then for each block, say x^i, we determine a set of substrings of y as candidate intervals such that one of the candidate substrings is close to the optimal substring x^i is matched to in the optimal matching. We call such a substring *approximately optimal candidate*. Then, if we know the edit distance (or a good approximation of edit distance) between each block and each of its candidate substring of y, we can run a dynamic programming to get an approximation of the actual edit distance.

In the following analysis, we fix an optimal alignment between x and y where the i-th block $x^i \triangleq x[l_i : r_i]$ is matched to substring $y[\alpha_i : \beta_i]$ such that the intervals $[\alpha_i, \beta_i]$'s are disjoint and span the entire length of y. By the assumption, we have $\mathsf{ED}(x, y) = \sum_{i=1}^{N} \mathsf{ED}(x^i, y[\alpha_i : \beta_i])$.

We now give the definition of (ε, Δ)-*approximately optimal candidate*.

Definition 2.2.1 ([34]). *We say an interval* $[\alpha', \beta']$ *is an* (ε, Δ)-approximately optimal candidate *of the block* $x^i = x[l_i : r_i]$ *if the following two conditions hold:*

$$\alpha_i \le \alpha' \le \alpha_i + \varepsilon\frac{\Delta}{b}$$

$$\beta_i - \varepsilon\frac{\Delta}{b} - \varepsilon\, \mathsf{ED}(x[l_i : r_i], y[\alpha_i : \beta_i]) \le \beta' \le \beta_i$$

We first show that if Δ is a good approximation of $\mathsf{ED}(x, y)$, and for each block that is not matched to a too large or a too small interval, we know the edit distance between it and one of its approximately optimal candidate, we can get a good approximation of $\mathsf{ED}(x, y)$. We put it formally in Lemma 2.2.1.

Lemma 2.2.1 (Implicit from [34]). *Let* $\varepsilon \in (0, 1)$ *(*ε *can be subconstant) and* $\varepsilon' = \varepsilon/10$. *Assume* $\mathsf{ED}(x, y) \le \Delta \le (1 + \varepsilon')\, \mathsf{ED}(x, y)$. *For each* $i \in [b]$, *let* (α_i', β_i') *be any* (ε', δ)-*approximately optimal candidate of* x^i. *If* $\varepsilon'|\alpha_i - \beta_i + 1| \le |x^i| \le 1/\varepsilon'|\alpha_i - \beta_i + 1|$, *let* $D_i' = |\alpha_i - \alpha_i'| + \mathsf{ED}(x^i, y[\alpha_i' : \beta_i']) + |\beta_i - \beta_i'|$. *Otherwise, let* $D_i' = |x^i| + |\alpha_i - \beta_i + 1|$.

Then

$$\mathsf{ED}(x,y) \le \sum_{i=1}^{b} D'_i \le (1+\varepsilon)\,\mathsf{ED}(x,y).$$

To make our work self-contained, we provide a proof in Appendix A.

We now show that, for each i and ε, Δ, without knowing the optimal alignment, we can pick a small set of candidate intervals such that one of the intervals is an (ε, Δ)-*approximately optimal candidate* for x^i.

That is, there exists a set of intervals $C^i_{\varepsilon,\Delta}$ with size $O(b \log n/\varepsilon^2)$ and one of the intervals in $C^i_{\varepsilon,\Delta}$ is an (ε, Δ)-*approximately optimal candidate* for x^i. The set $C^i_{\varepsilon,\Delta}$ can be found with the algorithm $\mathsf{CandidateSet}$ which is implicit from [34]. The algorithm takes six inputs : three integers n, m, and b, an interval (l_i, r_i), $\varepsilon \in (0,1)$, and $\Delta \le n$ and outputs set $C^i_{\varepsilon,\Delta}$. Here, n and m are the lengths of string x and y correpondingly. The pseudocode is given in algorithm 2.

Algorithm 2: $\mathsf{CandidateSet}$

```
Data: Integers $n$, $m$, and $b$, an interval $(l_i, r_i)$, $\varepsilon \in (0,1)$, and $\Delta \le n$
1: initialize $C$ to be an empty set.
2: for all $i' \in [l_i - \Delta - \varepsilon\frac{\Delta}{b}, l_i + \Delta + \varepsilon\frac{\Delta}{b}] \cap [m]$ such that $i'$ is a multiple of $\lceil \varepsilon\frac{\Delta}{b} \rceil$ do
       ▷ if $\lceil \varepsilon\frac{\Delta}{b} \rceil = 0$, we try every $i'$ in $[l_i - \Delta - 1, l_i + \Delta + 1] \cap [m]$.
3:     for all $j' = 0$ or $j' = \lceil (1+\varepsilon)^i \rceil$ for some integer $i \le \lceil \log_{1+\varepsilon}(m) \rceil$ do
           ▷ pick $O(\log_{1+\varepsilon} n)$ ending points.
4:         if $|x^i| - j' \ge \varepsilon|x^i|$ then
5:             add $(i', i' + |x^i| - 1 - j')$ to $C$.
6:         if $|x^i| + j' \le |x^i|/\varepsilon$ then
7:             add $(i', i' + |x^i| - 1 + j')$ to $C$.
8: return $C$.
```

Lemma 2.2.2 (Implicit from [34])**.** *If $\varepsilon m \le n \le \frac{1}{\varepsilon}m$, then $C^i_{\varepsilon,\Delta}$, the output of algorithm $\mathsf{CandidateSet}(n, m, b, (l_i, r_i), \varepsilon, \Delta)$, is of size $O(\frac{b\log n}{\varepsilon^2})$. For $x^i = x[l_i : r_i]$, if $\varepsilon|\alpha_i - \beta_i + 1| \le |x^i| \le 1/\varepsilon|\alpha_i - \beta_i + 1|$ and $\Delta \ge \mathsf{ED}(x,y)$, then one of the intervals in $C^i_{\varepsilon,\Delta}$ is an* (ε, Δ)-approximately optimal candidate *of x^i.*

A proof of Lemma 2.2.2 can be found in Appendix A.

Given the information of edit distances between x^i and each of intervals in $C^i_{\varepsilon,\Delta}$, we can run a simple dynamic programming algorithm EditDP to get an approximation of $\mathsf{ED}(x,y)$. EditDP takes six inputs, n, m, b, Δ, ε, and a two dimensional list M such that $M(i,(\alpha,\beta)) = \mathsf{ED}(x^i, y[\alpha:\beta])$ for each i and $(\alpha,\beta) \in C^i_{\varepsilon,\Delta}$. The pseudocode is given in algorithm 3.

Algorithm 3: EditDP

```
Data: three integers $n$, $m$, $b$, $\Delta \le n$, $\varepsilon \in (0,1)$, and a two dimensional list $M$
      such that $M(i,(\alpha,\beta)) = \mathsf{ED}(x^i, y[\alpha:\beta])$ for each $i$ and $(\alpha,\beta) \in C^i_{\varepsilon,\Delta}$.
1: let $C^i$ be the set of starting points of intervals in $C^i_{\varepsilon,\Delta}$ with no repetition for each
   $i \in [b]$.
2: for all $\alpha \in C^1$ do
3:     $A(0,\alpha-1) = \alpha-1$.     ▷ A is a two dimensional array for storing the
       intermediate results of the dynamic programming.
4: for all $i = 1$ to $b-1$ do
5:     for all $\alpha \in C^{i+1}$ do
6:         $A(i,\alpha-1) = \min\begin{cases}\min_{\alpha'\in C^i,\alpha'\le\alpha} A(i-1,\alpha'-1)+|\alpha-\alpha'|+|x^i| \\ \min_{\substack{(\alpha',\beta')\in C^i_{\varepsilon,\Delta} \\ \text{s.t. } \beta'\le\alpha-1}} A(i-1,\alpha'-1)+M(i,(\alpha',\beta'))+\alpha-1-\beta'\end{cases}$
7: $d = \min\begin{cases}\min_{\alpha'\in C^b} A(b-1,\alpha'-1)+|m-\alpha'|+|x^b| \\ \min_{\substack{(\alpha',\beta')\in C^b_{\varepsilon,\Delta} \\ \text{s.t. } \beta'\le m}} A(b-1,\alpha'-1)+M(b,(\alpha',\beta'))+m-\beta';\end{cases}$
8: return $d$
```

Lemma 2.2.3. *For any fixed ε, we let $\varepsilon' = \varepsilon/10$. Assume $\mathsf{ED}(x,y) \le \Delta \le (1+\varepsilon')\,\mathsf{ED}(x,y)$ and for every $(\alpha,\beta) \in C^i_{\varepsilon',\Delta}$, $M(i,(\alpha,\beta)) = \mathsf{ED}(x^i, y[\alpha:\beta])$, then $\mathsf{EditDP}(n,m,b,\varepsilon',\Delta,M)$ outputs a $(1+\varepsilon)$-approximation of $\mathsf{ED}(x,y)$ in $O(\frac{b^3\log n}{\varepsilon^3})$ time with $O(\frac{b}{\varepsilon}\log n)$ bits of space.*

Also, in the input, if we replace $M(i,(\alpha,\beta))$ with a $(1+\gamma)$ approximation of $\mathsf{ED}(x^i, y[\alpha:\beta])$, i.e.

$$\mathsf{ED}(x^i, y[\alpha:\beta]) \le M(i,(\alpha,\beta)) \le (1+\gamma)\,\mathsf{ED}(x^i, y[\alpha:\beta]),$$

then $\mathsf{EditDP}(n,m,b,\varepsilon',\Delta,M)$ outputs a $(1+\varepsilon)(1+\gamma)$-approximation of $\mathsf{ED}(x,y)$.

We note that the space complexity of algorithm 3, EditDP, is optimized. Let C^i be the set of starting points of intervals in $C^i_{\varepsilon,\Delta}$ with no repetition. Notice that when updating $A(i,\alpha)$, we only need the information of $A(i-1,\alpha'-1)$ for every $\alpha' \in C^i$. Thus, we can release the space used to store $A(i-2,\alpha''-1)$ for every $\alpha'' \in C^{i-1}$. Furthermore, for line 7, we only need the information of $A(i-1,\alpha-1)$ for every $\alpha \in C^{\cdot}$. From algorithm 2, we know that for each i, we pick at most b/ε points as the starting point of the candidate intervals. The size of C^i is at most b/ε. Since each element in A is an integer at most n, it can be stored with $O(\log n)$ bits of space. Thus, the space required is $O(\frac{b}{\varepsilon}\log n)$.

A full proof of Lemma 2.2.3 can be found in Appendix A.

Our space efficient algorithm recursively use the above ideas with carefully picked parameters.

In the following, b and ε are two parameters we will set later. We call our space-efficient approximation algorithm for edit distance ApproxED and give the pseudocode in Algorithm 4.

Algorithm 4: ApproxED: space efficient approximation of ED

```
Data: Two strings $x$ and $y$, parameters $b \le \sqrt{n}$ and $\varepsilon \in (0,1)$
 1: if $|x| \le b$ then
 2:    compute $\mathsf{ED}(x,y)$ exactly.
 3:    return $\mathsf{ED}(x,y)$.
 4: $\Lambda \leftarrow \infty$.
 5: set $n = |x|$ and $m = |y|$.
 6: divide $x$ into $b$ block each of length at most $\lceil n/b \rceil$ such that $x = x^1 \circ x^2 \circ \cdots \circ x^b$.
 7: for all $\Delta = 0$ or $\lceil (1+\varepsilon)^j \rceil$ for some integer $j$ and $\Delta \le \max\{|x|,|y|\}$ do
 8:    for $i = 1$ to $b$ do
 9:       for all $(a,b) \in \mathsf{CandidateSet}(n,m,(l_i,r_i),\varepsilon,\Delta)$ do
10:          $M(i,(a,b)) \leftarrow \mathsf{ApproxED}(x^i, y[a:b], b, \varepsilon)$.
11:    $\Lambda \leftarrow \min\{d, \mathsf{EditDP}(n,m,b,\varepsilon,\Delta,M)\}$.
12: return $\Lambda$.
```

We have the following result.

Lemma 2.2.4. *Given two strings $x, y \in \Sigma^n$, parameters $b \le \sqrt{n}$ and $\varepsilon \in (0,1)$, ApproxED(x,y,b,ε) outputs a $1+O(\varepsilon \log_b n)$-approximation of ED$(x,y)$ with* $O\left(\frac{b\log^2 n}{\varepsilon \log b}\right)$

bits of space in $(\frac{\varepsilon^2}{b\log b\log n}+\frac{\varepsilon^2}{\log^2 n})\cdot\left(O(\frac{b^2\log^2 n}{\varepsilon^3})\right)^{\lceil\log_b n\rceil}$ *time.*

Proof of Lemma 2.2.4. Algorithm 4 is recursive. We start from level one and every time $\mathsf{ApproxED}$ is called, we enter the next level. We say the largest level we will reach is the maximum depth of recursion. In the following, to avoid ambiguity, x and y denote the input strings at the first level where both string has length n.

Notice that the length of first input string at i-th level is at most $\frac{n}{b^{i-1}}$. The recursion terminates when the length of first input string is no larger than b. Thus the maximum depth of recursion is $\lceil\log_b n\rceil$. We denote the maximum depth of recursion by d.

We first show the correctness of our algorithm by prove the following claim.

Claim 2.2.1. *At the l-th level, the output is a $(1+10\varepsilon)^{d-l}$ approximation of the edit distance of its input strings.*

Proof. We prove this by induction on l from d to 1. For the base case $l=d$, we output the exact edit distance. The claim holds trivially.

Now, we assume the claim holds for the $(l+1)$-th level. At the level l, if the input string x has length no larger than b, we output the exact edit distance. The claim holds for level i. Otherwise, since we tried every $\Delta=\lceil(1+\varepsilon)^j\rceil$ for some integer j and $\Delta\le n+m$, one of Δ satisfies $\mathsf{ED}(x,y)\le\Delta\le(1+\varepsilon)\,\mathsf{ED}(x,y)$. Denote such a Δ by Δ_0. For $(a,b)\in C^i_{\varepsilon,delta}$ that $M(i,(a,b))$ is a $(1+10\varepsilon)^{d-(l+1)}$ approximation of $\mathsf{ED}(x^i,y[a:b])$ by the inductive hypothesis. By lemma 2.2.3, $\mathsf{EditDP}(n,m,b,\varepsilon,\Delta_0,M)$ outputs a $(1+10\varepsilon)(1+10\varepsilon)^{d-(l+1)}=(1+10\varepsilon)^{d-l}$ approximation of $\mathsf{ED}(x,y)$ when $\Delta=\Delta_0$. This proves the claim.

□

By the above claim, for the first level, our algorithm always output a $(1+10\varepsilon)^{d-1}=1+O(\varepsilon d)=1+O(\varepsilon\log_b n)$ approximation of $\mathsf{ED}(x,y)$.

We now turn to the space and time complexity. We can consider our recursion structure as a tree. The first level corresponds to the root of the recursion tree. Notice that we need to try $O(\log_{1+\varepsilon}(n))$ different Δ. For each Δ, we need to query the next level $O(b(\frac{b\log n}{\varepsilon^2}))$ times. This is because there are b blocks and for each block, we choose $O(\frac{b\log n}{\varepsilon^2})$ candidate intervals by lemma 2.2.2. Thus, the recursion tree has degree $O(\log_{1+\varepsilon}(n)\frac{b^2\log n}{\varepsilon^2}) = O(\frac{b^2\log^2 n}{\varepsilon^3})$ with depth $d \le \log_b n$.

Running the algorithm essentially has the same order as doing a depth first search on the recursion tree. At each level of the recursion, we only need to remember the information in one node. Thus, total space required is equal to the space needed for one inner node times the depth of recursion tree, plus the space needed for one leaf node.

For the leaf nodes, the first input string is of length at most b, we can compute the edit distance exactly with space $O(b\log n)$ bits of space.

For other nodes, the task is to run a dynamic programming, i.e. algorithm EditDP where we need to query the next level of recursion to get the matrix M. In algorithm EditDP, the input is a matrix M and we need to compute a matrix A. For matrices A, the rows are indexed by i from 0 to b and for the i-th row, the columns are indexed by the elements in set C^i. C^i is the set of starting points of intervals in $C^i_{\varepsilon,\Delta}$. By the proof of lemma 2.2.2, C^i is of size $O(\frac{b}{\varepsilon})$.

According to the proof of lemma 2.2.3, we can divide the dynamic programming into b steps and for each step, we update one row of A. When computing the i-th row of A, we only need to query the $i-1$-th row of A and the i-th row of M. Thus, the space used to remember previous rows of A can be reused. Also, we only query each element in the i-th row of M once, so we do not need to remember matrix M and this does not affect the time complexity. Thus, for each inner node of the recursion, we only need space enough for storing two rows of matrix A and each element of A is an integer no larger than n. The space for each inner node is bounded by $O(\frac{b}{\varepsilon}\log n)$ bits.

Thus, the space complexity of our algorithm is bounded by $O\Big(d\cdot \frac{b}{\varepsilon}\log n + b\log n\Big) = O\Big(\frac{b\log^2 n}{\varepsilon \log b}\Big)$ bits.

For time complexity, we denote the time used for computation at the i-th level by T_i (excluding the time used for running ApproxED at the i-level). The time complexity is bounded by the sum of time spent at each level. Denote the total running time by T, we have $T = \sum_{i=1}^{d} T_i$.

Once *SpaceEffientApproxED* is called at the $(i-1)$-th level, we enter the i-th level.

Each time we enter the i-th level, there are two possible cases. For the first case, the operation at the i-th level is calculating the exact edit distance with one of the input string has length at most b and the other string has length $O(b/\varepsilon)$. It takes $O(\frac{b^2}{\varepsilon})$ time.

Otherwise, we run EditDP for $O(\log_{1+\varepsilon} n) = O(\frac{\log n}{\varepsilon})$ times. By lemma 2.2.3, it takes $O(\frac{b^3 \log n}{\varepsilon^3})$ time.

Thus, each time we enter the i-th level, the time required at that level is bounded by $O(\frac{b^3\log^2 n}{\varepsilon^4})$. Since the recursion tree has degree $O(\frac{b^2\log^2 n}{\varepsilon^3})$, we enter the i-th level $(O(\frac{b^2\log^2 n}{\varepsilon^3}))^{i-1}$ times. Thus, $T_i = \frac{b^3\log^2 n}{\varepsilon^4}(O(\frac{b^2\log^2 n}{\varepsilon^3}))^{i-1}$.

For $1 \le i \le d-1$, T_i is bounded by $\frac{b^3\log^2 n}{\varepsilon^4}(O(\frac{b^2\log^2 n}{\varepsilon^3}))^{i-1}$. We have

$$\begin{aligned}\sum_{i=1}^{d-1} T_i &\le (d-1)T_{d-1}\\ &\le d\cdot \frac{b^3\log^2 n}{\varepsilon^4}\cdot \Big(O(\frac{b^2\log^2 n}{\varepsilon^3})\Big)^{d-2}\\ &= \frac{b^3\log^3 n}{\varepsilon^4 \log b}\cdot \Big(O(\frac{b^2\log^2 n}{\varepsilon^3})\Big)^{\lceil \log_b n\rceil -2}\\ &= \frac{\varepsilon^2}{b\log b\log n}\cdot \Big(O(\frac{b^2\log^2 n}{\varepsilon^3})\Big)^{\lceil \log_b n\rceil}.\end{aligned} \tag{2.2}$$

Also notice that at the d-th level, we always do the exact computation of edit distance, which takes $O(\frac{b^2}{\varepsilon})$ time. Thus

$$\begin{aligned} T_d &= \frac{b^2}{\varepsilon}\Big(O(\frac{b^2\log^2 n}{\varepsilon^3})\Big)^{d-1} \\ &= \frac{\varepsilon^2}{\log^2 n}\Big(O(\frac{b^2\log^2 n}{\varepsilon^3})\Big)^{d} \\ &= \frac{\varepsilon^2}{\log^2 n}\Big(O(\frac{b^2\log^2 n}{\varepsilon^3})\Big)^{\lceil \log_b n\rceil}. \end{aligned} \tag{2.3}$$

Combining 2.2 and 2.3. We know the running time is bounded by

$$(\frac{\varepsilon^2}{b\log b\log n}+\frac{\varepsilon^2}{\log^2 n})\cdot\Big(O(\frac{b^2\log^2 n}{\varepsilon^3})\Big)^{\lceil \log_b n\rceil}.$$

□

Lemma 2.2.5. *Given two strings x and y, both of length n, there is a deterministic algorithm that outputs a $1+O(\frac{1}{\log\log n})$ approximation of $\mathsf{ED}(x,y)$ with $O(\frac{\log^4 n}{\log\log n})$ bits of space in $O(n^{7+o(1)})$ time.*

Proof of Lemma 2.2.5. Let $b=\log n$ and $\varepsilon=\frac{1}{\log n}$. By Lemma 2.2.4, the space complexity is

$$O\Big(\frac{b\log^2 n}{\varepsilon\log b}\Big)=O\Big(\frac{\log^4 n}{\log\log n}\Big).$$

The running time is

$$\begin{aligned} &(\frac{\varepsilon^2}{b\log b\log n}+\frac{\varepsilon^2}{\log^2 n})\cdot\Big(O(\frac{b^2\log^2 n}{\varepsilon^3})\Big)^{\lceil \log_b n\rceil} \\ =&\frac{1}{\log^4 n}(O(\log^7 n))^{\lceil\frac{\log n}{\log\log n}\rceil} \\ =&\frac{1}{\log^4 n}(2^{7\log\log n+O(1)})^{\frac{\log n}{\log\log n}+1} \\ =&\frac{1}{\log^4 n}O(n^{7+o(1)}) \\ =&O(n^{7+o(1)}). \end{aligned}$$

□

Lemma 2.2.6. *Given two strings x and y, both of length n, for any fixed constant $\varepsilon\in(0,1)$, $\delta\in(0,\frac{1}{2})$, there is a deterministic algorithm that outputs a $1+\varepsilon$ approximation of $\mathsf{ED}(x,y)$ with $\tilde{O}_{\varepsilon,\delta}(n^\delta)$ bits of space in $\tilde{O}_{\varepsilon,\delta}(n^2)$ time*

Proof of Lemma 2.2.6. Let $b = n^\delta$ and pick ε' to be a constant sufficiently smaller than ε. We run algorithm ApproxED with inputs x, y, b, and ε'. Then Theorem 2.2.6 is a direct result of Lemma 2.2.4. □

Theorem 2.1 is a direct result of Lemma 2.2.5 and Lemma 2.2.6.

2.3 Space Efficient Approximation for LIS

We now present our space-efficient algorithms for LIS. In this Section, we assume the alphabet Σ is an ordered set. We call our main algorithm ApproxLIS and give the pseudocode in Algorithm 5. We also introduce a slightly modified version called ApproxLISBound. ApproxLISBound takes an additional input l, which is an integer at most n. We want to guarantee that, if the input sequence x has an increasing subsequence of length l ending with $\alpha \in \Sigma$, then ApproxLISBound(x, b, ε, l) can detect an increasing subsequence with length close to l, and ending with some symbol in Σ at most α.

ApproxLISBound is similar to ApproxLIS with only a few differences. First, at line 11 of algorithm ApproxLIS, we always require k to be at most l. That is, we let $k = \min\{l, \max\{k, s+d\}\}$. Second, instead of output $\max S$, we output the whole set S and list Q (The streaming algorithm from [14] also maintains set S and list Q). We omit the pseudocode for ApproxLISBound.

Lemma 2.3.1. *Given a sequence $x \in \Sigma^n$ and two parameters $b \le \sqrt{n}$ and $\varepsilon \in (0,1)$, ApproxLIS(x, b, ε) computes a $(1 - 3\log_b(n)\varepsilon)$ approximation of LIS(x) with $O\Big(\frac{b \log^2 n}{\varepsilon \log b}\Big)$ bits of space in $\Big(O(\frac{b^2}{\varepsilon^2}\log n)\Big)^{\lceil \log_b n \rceil - 2} \cdot \Big(b^2 \log n + \frac{b \log n}{\varepsilon \log b}\Big)$ time.*

Proof of Lemma 2.3.1. ApproxLIS is a recursive algorithm. We start from level one and every time ApproxLIS or ApproxLISBound is called, we enter the next level. Assume the input string at the first level has length n. Notice that except the last level, we

Algorithm 5: ApproxLIS (ApproxLISBound): space efficient approximation of LIS

```
Data: A string $x$ , parameters $b$ and $\varepsilon$. And an additional parameter $l$ for
      ApproxLISBound
 1: if $|x| \le b^2$ then
 2:    compute an $(1-\varepsilon)$-approximation of $\mathsf{LIS}(x)$ with the streaming algorithm from
       [14] using $O(\frac{b}{\varepsilon}\log n)$ space. (For ApproxLISBound, we only consider LIS of
       length at most $l$.)
 3:    return
 4: divide $x$ evenly into $b$ blocks such that $x = x^1 \circ x^2 \circ \cdots \circ x^b$.      ▷ $|x^i| \le \lceil n/b \rceil$
 5: initialize $S = \{0\}$ and $Q[0] = -\infty$.
 6: for $i = 1$ to $b$ do
 7:    $k = 0$.
 8:    for all $s \in S$ do
 9:       let $z$ be the subsequence of $x^i$ by only considering the elements larger than
          $Q[s]$.
10:       $d = \mathsf{ApproxLIS}(z, b, \varepsilon)$.
11:       $k = \max\{k, s+d\}$. (For ApproxLISBound, we let $k = \min\{l, \max\{k, s+d\}\}$)
          .
12:    if $k \le b/\varepsilon$ then
13:       let $S' = \{0, 1, 2, \ldots, k\}$.
14:    else
15:       let $S' = \{0, \frac{\varepsilon}{b}k, 2\frac{\varepsilon}{b}k, \ldots, k\}$.      ▷ *evenly pick $b/\varepsilon + 1$ integers from 0 to $k$
          (including 0 and $k$).*
16:    $Q'[s] = \infty$ for all $s' \in S'$ except $Q'[0] = -\infty$.
17:    for all $s \in S$ do
18:       let $z$ be the subsequence of $x^i$ by only considering the elements larger than
          $Q[s]$.
19:       for all $l = 1, 1+\varepsilon, (1+\varepsilon)^2, \ldots, k-s$ do
20:          $\tilde{S}, \tilde{Q} \leftarrow \mathsf{ApproxLISBound}(z, b, \varepsilon, l)$ .
21:          for each $s' \in S'$ such that $s \le s' \le s+l$, let $\tilde{s}$ be the smallest element in $\tilde{S}$
             that is larger than $s'-s$ and set $Q'[s'] = \min\{\tilde{Q}[\tilde{s}], Q'[s']\}$.
22:    $S \leftarrow S'$, $Q \leftarrow Q'$.
23: return $\max S$. (for ApproxLISBound, we return the sets $S$ and $Q$)
```

always divide the string evenly into b blocks. So the length of input string at the i-th level is bounded by $\lceil \frac{n}{b^{i-1}} \rceil$. The recursion stops when the input string has length no larger than b^2. Thus, the depth of recursion is at most $\lceil \log_b \rceil n - 1$. In the following, we denote the depth of recursion by d.

To prove the correctness of our algorithm, we show the following claim.

Claim 2.3.1. *At the i-th level of recursion, let x denote the input string at this level. ApproxLIS(x, b, ε) outputs a $(1 - 3(d-i)\varepsilon)$ approximationg of LIS(x). For any l, if there is an increasing subsequence of x with length l and ending with $\alpha \in \Sigma$, then ApproxLISBound(x, b, ε, l) outputs a set S and a list Q, such that there is an element $s \in S$ with*

$$\Big(1 - 3(d-i)\varepsilon\Big) \cdot l \leq s \leq l$$

and $Q[s] \leq \alpha$.

Proof of Claim 2.3.1. For simplicity, we abuse the notation a little by denoting the input string at i-th level by x. The proof is by induction on i from d to 1.

For the base case $i = d$, the input string x has length at most b^2. We run the $(1-\varepsilon)$ approximation algorithm from [14]. The claim holds trivially by the correctness of that algorithm.

Assume the claim holds for $i+1$-th level for $1 \leq i \leq d-1$. We now prove it also holds for i-th level. Let x be the input string at the i-th level. We start by showing the correctness of ApproxLIS.

For our analysis, let τ be one of the longest increasing subsequence of x. τ can be divided into b parts such that $\tau = \tau^1 \circ \tau^2 \circ \cdots \circ \tau^b$ and τ^i lies in x^i. We define the following variables.

α_i is the first symbol of τ^i (if τ^i is not empty).

β_i is the last symbol of τ^i (if τ^i is not empty).

$d_i = |\tau^i|$ is the length of τ_i.

$\gamma^i = \tau^1 \circ \tau^2 \circ \cdots \circ \tau^i$ is the concatenation of the first i blocks in τ.

$h_i = \sum_{j=1}^{i} d_j = |\gamma^i|$ is the length of γ^i.

In the PatienceSorting algorithm, we initialize a list P with n elements such that $P[i] = \infty$ for all $i \in [n]$, and then scans the input sequence x from left to right. When reading a new symbol x_i, we find the smallest index l such that $P[l] \geq x_i$ and set $P[l] = x_i$. After processing the string x, for each i such that $P[i] < \infty$, we know $P[i]$ is the smallest possible character such there is an increasing subsequence in x of length i ending with $P[i]$. Finally the algorithm returns the largest index l such that $P[l] < \infty$. In the following, we let P be the list we get after running PatienceSorting with input x.

P' is the list "interpolated" by Q such that $P'[i] = Q[j]$ for the smallest $j \geq i$ that lies in S. If no such j exist, set $P'[i] = \infty$. We denote the set S and list Q after processing the block x^t (the t-th outer loop) by S_t and Q_t and the largest element in S_t by k_t. Correspondingly, P'_t is the list P' after processing the t-th block x^t and P_t is the list after running PatienceSorting with input $x^1 \circ x^2 \circ \cdots \circ x^t$.

Since τ is a longest increasing subsequence, without loss of generality, we can assume $P_t[h_t] = \beta_t$ (if τ^t is not empty) for each t from 1 to b. This is because, if $P_t[h_t] < \beta_t$, we can replace γ^t with another increasing subsequence of $x^1 \circ x^2 \circ \cdots \circ x^t$ with length h_t and ends with $P_t[h_t]$. On the other hand, we must have $P_t[h_t] \leq \beta_t$ since γ^t is an increasing subsequence of $x^1 \circ x^2 \circ \cdots \circ x^t$ with length h_t.

We also assume that $P_t[h_t] = P_{t+1}[h_t]$ if τ^t is an empty string ($h_t = h_{t+1}$). Since if not, we can replace γ^{t+1} with another increasing substring with τ^t not empty.

We have the following claim.

Claim 2.3.2. *For each $t \in [b]$, we have*

$$P'_t\left[\Big(1 - 3\big(d - (i+1)\big)\varepsilon - \varepsilon\Big)h_t - \frac{2\varepsilon t}{b}k_t\right] \leq P_t[h_t].$$

Proof of Claim 2.3.2. We prove this by induction on t.

For the base case $t = 1$, if $d_1 = 0$, then $h_1 = 0$. $P'_1[-2\varepsilon t n^{-\frac{1}{d+1}} k_t]$ is not defined, we assume without loss of generality that $P'_1[\theta] = -\infty$ if $\theta \le 0$. Since $P_1[0]$ and $P'_1[0]$ are both special symbol $-\infty$, the claim holds.

If $d_1 > 0$, we have $d_1 = h_1$. Let l be the largest number such that $l = (1+\varepsilon)^j$ for some integer j and $l \le d_1$. We have

$$\frac{1}{1+\varepsilon} \cdot d_1 \le l \le d_1.$$

Let $\tilde{S}, \tilde{Q}$ be the output of $\mathsf{ApproxLISBound}(x^1, b, \varepsilon, l)$. By our assumption on the correctness of Claim 2.3.1 on $i+1$ recursive level, there exist an $\tilde{s} \in \tilde{S}$ such that

$$\begin{aligned}\tilde{s} &\ge \Big(1 - 3(d-(i+1))\varepsilon\Big) \cdot l \\ &\ge \frac{1-3(d-(i+1)\varepsilon)}{1+\varepsilon} d_1 \\ &\ge (1 - 3(d-(i+1))\varepsilon - \varepsilon) d_1\end{aligned} \tag{2.4}$$

and

$$\tilde{Q}[\tilde{s}] \le P_1[l] \le P_1[h_1] = \beta_1.$$

By the choice of S_1 (line 21 of algorithm 5), we know there is an $s \in S_1$ such that $\tilde{s} - \frac{\varepsilon}{b} \cdot k_1 \le s \le \tilde{s}$ and $Q_1[s] \le \tilde{Q}[\tilde{s}]$.

Combining 2.4, we have

$$\begin{aligned}&P'_1\left[\left(1 - 3\Big(d-(i+1)\Big)\varepsilon - \varepsilon\right)h_1 - \frac{2\varepsilon}{b}k_1\right] \\ \le& P'_1[\tilde{s} - \frac{2\varepsilon}{b}k_1] \le P_1[s] \le \tilde{Q}[\tilde{s}] \\ \le& P_1[h_1]\end{aligned} \tag{2.5}$$

This proved the base case of $t = 1$.

Now we assume the claim holds for some fixed integer $t - 1 \le b$, we show it also holds for t. If τ^t is an empty string, we have $h_t = h_{t-1}$ and $P_{t-1}[h_t] = P_t[h_t]$. Since

$k_t \geq k_{t-1}$, we have

$$\begin{aligned}
&P'_t\left[\Big(1-3\big(d-(i+1)\big)\varepsilon-\varepsilon\Big)h_t-\frac{2\varepsilon t}{b}k_t\right]\\
\leq&P'_t\left[\Big(1-3\big(d-(i+1)\big)\varepsilon-\varepsilon\Big)h_t-\frac{2\varepsilon t}{b}k_{t-1}\right]\\
\leq&P'_{t-1}\left[\Big(1-3\big(d-(i+1)\big)\varepsilon-\varepsilon\Big)h_t-\frac{2\varepsilon t}{b}k_{t-1}\right]\\
\leq&P_{t-1}[h_t]=P_t[h_t].
\end{aligned}$$

Thus, the claim holds for the case when τ^t is an empty string.

If $d_t > 0$ (τ^t is not empty), by the assumption that

$$P'_{t-1}\left[\Big(1-3\big(d-(i+1)\big)\varepsilon-\varepsilon\Big)h_{t-1}-\frac{2\varepsilon(t-1)}{b}k_{t-1}\right]\leq P_{t-1}[h_{t-1}], \tag{2.6}$$

we know there is an $s_a \in S_{t-1}$ such that

$$s_a \geq \Big(1-3(d-(i+1))\varepsilon-\varepsilon\Big)h_{t-1}-\frac{2\varepsilon(t-1)}{b}k_{t-1}-\frac{\varepsilon}{b}k_{t-1}, \tag{2.7}$$

and

$$s_a \leq \Big(1-3(d-(i+1))\varepsilon-\varepsilon\Big)h_{t-1}-\frac{2\varepsilon(t-1)}{b}k_{t-1}. \tag{2.8}$$

Also,

$$Q_{t-1}[s_a] \leq P'_{t-1}[(1-3(d-(i+1))\varepsilon-\varepsilon)h_{t-1}-2(t-1)\frac{\varepsilon}{b}k_{t-1}].$$

Similarly, we let l be the largest number such that $l=(1+\varepsilon)^j$ for some integer j and $l \leq d_t$. That is

$$\frac{1}{1+\varepsilon}d_t \leq l \leq d_t$$

We run $\mathsf{ApproxLISBound}(x^t,\varepsilon,l)$ to get $\tilde{S}$ and $\tilde{Q}$. By our assumption on the correctness of $\mathsf{ApproxLISBound}$ on the $(i+1)$-th level, there exist an $\tilde{s} \in \tilde{S}$ such that

$$\begin{aligned}
\tilde{s} &\geq \Big(1-3(d-(i+1))\varepsilon\Big)l\\
&\geq \Big(1-3(d-(i+1))\varepsilon\Big)\frac{d_t}{1+\varepsilon}\\
&\geq \Big(1-3(d-(i+1))\varepsilon-\varepsilon\Big)d_t
\end{aligned} \tag{2.9}$$

and

$$\tilde{Q}[\tilde{s}] \leq P_t[s_a + l] \leq P_t[h_t] = \beta_t.$$

Let s_b be the largest element in S_t such that $s_b \leq s_a + \tilde{s}$. We know $Q_t[s_b] \leq \tilde{Q}[\tilde{s}] \leq P_t[h_t]$ by the updating rule at line 21 of algorithm 5. By the choice of set S_t and combining 2.7, 2.8, 2.9, we have

$$\begin{aligned} s_b &\geq s_a + \tilde{s} - \frac{\varepsilon}{b} k_t \\ &\geq \Big(1 - 3(d-(i+1))\varepsilon - \varepsilon\Big)(h_{t-1} + d_t) - \frac{2\varepsilon(t-1)}{b} k_{t-1} - \frac{\varepsilon}{b} k_{t-1} - \frac{\varepsilon}{b} k_t \\ &\geq \Big(1 - 3\Big(d - (i+1)\Big)\varepsilon - \varepsilon\Big) h_t - \frac{2\varepsilon t}{b} k_t \end{aligned}$$

The last inequality is from the fact that $h_t = h_{t-1} + d_t$ and $k_t \geq k_{t-1}$. Since $P'_t[s_b] \leq P_t[h_t]$, we have shown that

$$P'_t\left[\Big(1 - 3\Big(d - (i+1)\Big)\varepsilon - \varepsilon\Big) h_t - \frac{2\varepsilon t}{b} k_t\right] \leq P_t[h_t]$$

This finishes our proof of Claim 2.3.2. □

In Claim 2.3.2, when $t = b$, since $h_b \geq k_b$, we have

$$\begin{aligned} &P'_b\left[\Big(1 - 3(d-i)\varepsilon\Big) h_b\right] \\ \leq &P'_b\left[\Big(1 - 3\Big(d - (i+1)\Big)\varepsilon - \varepsilon\Big) h_b - 2\varepsilon k_b\right] \\ \leq &P_b[h_b] \end{aligned}$$

Thus, the output of $\mathsf{ApproxLIS}$ at i-th level is at least a $\Big(1 - 3(d-i)\varepsilon\Big)$ approximation of $\mathsf{LIS}(x)$.

It remains to show the correctness of $\mathsf{ApproxLISBound}$ at level i.

The analysis is essentially the same except now we replace the longest increasing subsequence τ with an increacing subsequence of length l ending with the smallest possible symbol in Σ.

Assume τ ends with $\sigma \in \Sigma$, we know τ is the longest increasing subsequence ending with σ. Since otherwise, we can find some $\sigma' < \sigma$ such that there is an increasing subsequence of length l ending with σ'.

Thus, we can similarly define of $\alpha_i, \beta_i, d_i, \gamma_i, h_i$ for $i \in [b]$. We can assume $P_t[h_t] = \beta_t$ if τ^t is not empty and $P_{t-1}[h_{t-1}] = P_t[h_t]$ otherwise. The remaining analysis are mostly the same. We omit the details.

□

By Claim 2.3.1, at the first level, the output is a $(1 - 3d\varepsilon)$ approximation of the length of LIS. Thus, $\mathsf{ApproxLIS}(x, b, \varepsilon)$ outputs a $\Big(1 - 3\log_b(n)\varepsilon\Big)$ approximation of $\mathsf{LIS}(x)$.

We now turn to time and space complexity. Our algorithm ApproxLIS is recursive and calls itself or ApproxLISBound, which has the same recursive structure. Each time ApproxLIS and ApproxLISBound is called at $(i-1)$-th recursive level, we enter the i-th level. When we enter the i-th level, if the input sequence has length larger than b, we need to call ApproxLIS $O(b|S|)$ times and ApproxLISBound $O(b|S|\log_{1+\varepsilon} n)$ times. The recursion tree has degree $O(b|S|\log_{1+\varepsilon} n) = O(\frac{b^2}{\varepsilon^2}\log n)$

The order of computation is the same as doing a depth first search on the recursion tree. At each level of the recursion, we only need to remember the information in one node. For the leaf nodes, we run streaming algorithm from [14] on a string of length at most b^2. It takes $O(\frac{b}{\varepsilon}\log n)$ space. For the inner nodes, we need to maintain a set $S \subseteq [n]$ and a list Q, both has size $O(\frac{b}{\varepsilon})$. Since each element in the set S or list Q takes $O(\log n)$ space. The space needed for one inner node is $O(\frac{b}{\varepsilon}\log n)$.

Since the depth of recursion is $d \leq \lceil \log_b \rceil n - 1$, the total space for ApproxLIS is bounded by $O\Big(d\frac{b}{\varepsilon}\log n\Big) = O\Big(\frac{b\log^2 n}{\varepsilon \log b}\Big)$.

For the time complexity, we first consider the time used within one level (excluding the time used for running ApproxLIS and ApproxLISBound). We denote the time used within i-th level by T_i and the total running time by T. We have $T = \sum_i^d T_i$.

For each node of the recursion tree, if the length of input string is at most b^2 (corresponding to the leaf nodes of the recursion tree), we run the streaming algorithm from [14]. It takes time $O(b^2 \log b)$. Since the recursion tree has degree $O(\frac{b^2}{\varepsilon^2} \log n)$ and depth d, the number of nodes at the bottom level is bounded by $\Big(O(\frac{b^2}{\varepsilon^2} \log n)\Big)^{d-1}$. Since $d \leq \lceil \log_b \rceil n - 1$, we have

$$T_d = \Big(O(\frac{b^2}{\varepsilon^2} \log n)\Big)^{\lceil \log_b n \rceil - 2} \cdot b^2 \log n. \tag{2.10}$$

If the length of input string is more than b^2, the time is then dominated by the operations at line 21 of algorithm 5. Since the size of S and $\tilde{S}$ are both at most $\frac{b}{\varepsilon}$ and we try at most $\log_{1+\varepsilon} n$ different l, it takes $O(b|S|^2 \log_{1+\varepsilon} n) = O(\frac{b^3}{\varepsilon^3} \log n)$ time. Also, the number of nodes in i-th level is at most $O\Big((\frac{b^2}{\varepsilon^2} \log n)^{i-1}\Big)$. We know

$$T_i = O\Big((\frac{b^2}{\varepsilon^2} \log n)^{i-1} \cdot \frac{b^3}{\varepsilon^3} \log n\Big).$$

Thus

$$\begin{aligned} \sum_{i=1}^{d-1} T_i &\leq (d-1) T_{d-1} \\ &\leq d \cdot (O(\frac{b^2}{\varepsilon^2} \log n))^{d-2} \cdot \frac{b^3}{\varepsilon^3} \log n) \\ &= \Big(O(\frac{b^2}{\varepsilon^2} \log n)\Big)^{\lceil \log_b n \rceil - 2} \cdot \frac{b \log n}{\varepsilon \log b} \end{aligned} \tag{2.11}$$

Combining 2.10 and 2.11 and $d \leq \log_b n - 1$, we know

$$T = \Big(O(\frac{b^2}{\varepsilon^2} \log n)\Big)^{\lceil \log_b n \rceil - 2} \cdot (b^2 \log n + \frac{b \log n}{\varepsilon \log b}) \tag{2.12}$$

□

Lemma 2.3.2. *Given a string $x \in \Sigma^n$, there is a deterministic algorithm that computes a $1 - O(\frac{1}{\log \log n})$ approximation of $\mathsf{LIS}(x)$ with $O(\frac{\log^4 n}{\log \log n})$ bits of space in $n^{5+o(1)}$ time.*

Proof of Lemma 2.3.2. Let $b = \log n$ and $\varepsilon = \frac{1}{\log n}$. Similar to Theorem 2.2.5, it is a direct result of Lemma 2.3.1. □

Lemma 2.3.3. *Given a string $x \in \Sigma^n$, for any constant $\delta \in (0, \frac{1}{2})$ such that $\frac{1}{\delta}$ is an integer and $\varepsilon \in (0,1)$, there is a deterministic algorithm that computes a $1-\varepsilon$ approximation of $\mathsf{LIS}(x)$ with $\tilde{O}_{\varepsilon,\delta}(n^{\delta})$ bits of space in $\tilde{O}_{\varepsilon,\delta}(n^{2-2\delta})$ time.*

Proof of Lemma 2.3.3. Let $b = n^{\delta}$ and pick ε' to be a constant sufficiently smaller than ε. Notice that when $\frac{1}{\delta}$ is an integer, we have $\lceil \log_b n \rceil = \lceil \frac{1}{\delta} \rceil = \frac{1}{\delta}$. We run $\mathsf{ApproxLIS}(x, b, \varepsilon')$. Then Theorem 2.3.3 is a direct result of Lemma 2.3.1. □

Theorem 2.3 is a direct result of Lemma 2.3.2 and Lemma 2.3.3.

Output the Sequence We can actually modify our algorithm to output the increasing subsequence we found (not only the length) at the cost of increased running time. We now show how it works.

We have the following result.

Lemma 2.3.4. *Given a sequence $x \in \Sigma^n$ and two parameters $b \le \sqrt{n}$ and $\varepsilon \in (0,1)$, setting $l = n$ and $\bar{s}$ be the output of $\mathsf{ApproxLIS}(x, b, \varepsilon)$. Then $\mathsf{LISSequence}(x, b, \varepsilon, l, \bar{s})$ outputs an increasing subsequence of x with length at least $(1 - 3\log_b(n)\varepsilon)\,\mathsf{LIS}(x)$ with $O\Big(\frac{b \log^2 n}{\varepsilon \log b}\Big)$ bits of space in $\Big(O(\frac{b^2}{\varepsilon^2} \log n)\Big)^{\lceil \log_b n \rceil - 1} \cdot \Big(b^3 \log n + \frac{b^2 \log n}{\varepsilon \log b}\Big)$ time.*

Proof of Lemma 2.3.4. The algorithm $\mathsf{LISSequence}$ is again recursive. At the first level, we set $l = n$ and let $\bar{s}$ be the output of $\mathsf{ApproxLIS}(x, b, \varepsilon)$. Notice that when $l = n$, $\mathsf{ApproxLISBound}$ is exactly the same as $\mathsf{ApproxLIS}$ since the length of $\mathsf{LIS}(x)$ can not be larger than n.

Let S_b and Q_b be the set and the list we get after running $\mathsf{ApproxLIS}$ (or equivalently, $\mathsf{ApproxLISBound}$ with $l = n$) with inputs x, b, ε. Let $s_b = \bar{s} = \max S_b$. We know s_b is at least $(1 - 3\log_b(n)\varepsilon)\,\mathsf{LIS}(x)$ and there is an increasing subsequence of x with length

Algorithm 6: LISSequence: output the subsequence detected by ApproxLIS

```
Data: A string $x$ ,parameters $b$, $\varepsilon$, $l$, $\bar{s}$.
 1: if $|x| \leq b$ then
 2:     output the exact longest increasing subsequence with $O(b \log n)$ bits of space in
        $O(b \log n)$ time (see [29] for example).
 3: divide $x$ evenly into $b$ blocks such that $x = x^1 \circ x^2 \circ \cdots \circ x^b$.
 4: compute $S_b$ and $Q_b$ by running ApproxLISBound$(x, b, \varepsilon, l)$  ▷ $S_i$ and $Q_i$ are the set
    $S$ and list $Q$ after $i$-th outer loop of ApproxLISBound .
 5: set $B$ to be a list with $B[0] = -\infty$.
 6: set $B[b] = Q_b[\bar{s}]$ and $s_b = \bar{s}$.                    ▷ we guarantee that $\bar{s} \in S_b$
 7: for $i = b - 1$ to $1$ do
 8:     release the space used for storing $S_{i+1}$, and $Q_{i+1}$.
 9:     compute $S_i$, $Q_i$ by running ApproxLISBound$(x, \varepsilon, l)$.
10:     for all $s \in S_i$ such that $s \leq s_{i+1}$ do
11:         let $z$ be the subsequence of $x^i$ by only considering the elements larger than
            $Q[s]$.
12:         for all $\tilde{l} = 1, 1 + \varepsilon, (1 + \varepsilon)^2, \ldots, k - s$ do
13:             $\tilde{S}, \tilde{Q} \leftarrow$ ApproxLISBound$(z, b, \varepsilon, \tilde{l})$.
14:             if there is an $\tilde{s} \in \tilde{S}$ such that $\tilde{s} + s \geq s_{i+1}$ and $B[i+1] = \tilde{Q}[\tilde{s}]$, we set
                $B[i] = Q_i[s]$, $l_i = \tilde{l}$, $s_i = s$, $\bar{s}_i = \tilde{s}$ and continue the loop at line 7 .
15: for $i = 1$ to $b$ do
16:     let $z$ be the subsequence of $x^i$ ignoring every element no larger than $B[i-1]$.
17:     LISSequence$(z, b, \varepsilon, l_i, \bar{s}_i)$.
```

s_b ending with $Q_b[s_b]$ by Lemma 2.3.1. Although we have detected such a sequence but we only know its length and the last symbol. Our goal is to output this sequence. For convenience, we denote this sequence by ρ.

ρ can be divided into b blocks such that $\rho = \rho^1 \circ \rho^2 \circ \cdots \circ \rho^b$ where ρ^i lies in x^i. Our goal is to recover a list B such that the last character of ρ^i is $B[i]$ (if ρ^i is not empty). We set $B[b] = Q_b[s_b]$ since ρ_b ending with $Q_b[s_b]$. Then we compute $B[i]$ from $i = b - 1$ to 1 by running ApproxLISBound multiple times.

We can do the following. Assume we already know s_{i+1} and $B[i+1]$, which means $\rho^1 \circ \rho^2 \cdots \circ \rho^{i+1}$ has length at least s_{i+1} and the last character is $B[i+1]$. By line 21 of ApproxLIS (ApproxLISBound), we know there must exist some $s_i \in S_i$ and $l_i = (1+\varepsilon)^j$ for some integer j, such that ApproxLISBound(z, b, ε, l_i) (here, z is the subsequence of x^i ignoring all symbols no larger than $Q_i[s_i]$) will detect an increasing subsequence of z whose last character is $B[i+1]$ and has length $\bar{s}_i \geq s_{i+1} - s'$. We can find such a s_i and l_i by trying every $s \in S_i$ and l. After this, we set $B[i] = Q_i[s_i]$.

Once we have computed B, we know the first element of ρ^i is larger than $B[i-1]$ and the last element is at most $B[i]$. Also, for each i from 1 to b, let z be the subsequence of x^i ignoring all symbols no larger than $B[i-1]$. Let $\tilde{S}$ and $\tilde{Q}$ be the output of ApproxLISBound(z, b, ε, l_i). We have $\bar{s}_i \in \tilde{S}$ and $\tilde{Q}[\bar{s}_i] = B[i]$. We can recursively use LISSequence with $l = l_i$ and $\bar{s} = \bar{s}_i$ to retrieve the subsequence ρ^i. The algorithm LISSequence follows the steps of ApproxLIS and ApproxLISBound. Since both algorithms are deterministic, we are guaranteed to output a sequence with length at least the output of ApproxLIS.

For the space complexity, LISSequence is also a recursive algorithm. It needs to call it self b times. We start from the first level, every time we enter the next level, the length of input string is decreased by a factor of b. Thus, the recursion tree is of degree b with depth at most $\lceil \log_b n \rceil$. The computation is in the same order as depth-first search on the recursion tree. We only need to remember the information of one node in each level. For the leaf nodes, we do the exact compution with $O(b \log n)$ bits of space.

In each inner node, we maintain a list B of size b. It takes $O(b \log n)$ space. We also run ApproxLIS and ApproxLISBound multiple times. By Lemma 2.3.1, this takes $O(\frac{b \log^2 n}{\varepsilon \log b})$ bits of space. The space used for running ApproxLIS and ApproxLISBound can be reused. Thus, the space complexity of algorithm LISSequence is $O(\log_b n \cdot \lceil b \log n \rceil + \frac{b \log^2 n}{\varepsilon \log b}) = O(\frac{b \log^2 n}{\varepsilon \log b})$.

For the time complexity, the running time can be divided into two parts: the time used for running ApproxLIS and ApproxLISBound, and the time used by LISSequence itself.

We start with the time used by LISSequence (assuming ApproxLIS and ApproxLISBound are oracles and can get results in constant time). LISSequence is recursive. Its recursion tree has degree b and depth at most $\lceil \log_b n \rceil$. For each leaf node, it takes $O(b \log n)$ time and the number of leaf nodes is bounded by $b^d = b^{\lceil \log_b n \rceil - 1}$. For each inner node, it computes list B. The time is dominated by the operations at line 14 of algorithm 6. It takes $O(b|S||\tilde{S}| \log_{1+\varepsilon} n) = O(\frac{b^3}{\varepsilon^3} \log n)$ time. Also, the number of inner nodes is bounded by $O(\log_b n b^{\lceil \log_b n \rceil - 2})$.

Thus, the total running time used by LISSequence itself is bounded by

$$O(\log_b n \cdot b^{\lceil \log_b n \rceil - 2} \cdot \frac{b^3}{\varepsilon^3} \log n + b \log n \cdot b^{\lceil \log_b n \rceil - 1}) = O\Big(\frac{b \log^n}{\varepsilon^3} b^{\lceil \log_b n \rceil}\Big) \tag{2.13}$$

Now we compute the time used for running ApproxLIS and ApproxLISBound. Since b and ε are fixed parameters. Let $f(m)$ denote time required for running ApproxLIS or ApproxLISBound once with input string length m.

Notice that when we compute ApproxLIS or ApproxLISBound with input string length n, we need to compute ApproxLISBound with input string length $\frac{m}{b}$ at most $O(\frac{b^2}{\varepsilon^2} \log n)$ times. Thus, we have

$$(\frac{b^2}{\varepsilon^2}\log n)\cdot f(\frac{m}{b}) \le f(m) \tag{2.14}$$

By Lemma 2.3.1, we know

$$f(n) = \Big(O(\frac{b^2}{\varepsilon^2}\log n)\Big)^{\lceil \log_b n\rceil - 2}\cdot (b^2\log n + \frac{b\log n}{\varepsilon \log b}) \tag{2.15}$$

At the first recursive level of $\mathsf{LISSequence}$, we need to run $\mathsf{ApproxLIS}$ b times with input string length n and $\mathsf{ApproxLISBound}$ $O(b|S|\log_{1+\varepsilon} n) = O(\frac{b^2}{\varepsilon^2}\log n)$ times with input string length $\frac{n}{b}$.

Thus, the time for running $\mathsf{ApproxLIS}$ and $\mathsf{ApproxLISBound}$ at first recursive level is bounded by $O(bf(n))$.

At the i-th recursive level, we need to run $\mathsf{ApproxLIS}$ b^i times with input string length $\frac{n}{b^{i-1}}$ and $\mathsf{ApproxLISBound}$ $O(b^{i-1}b|S|\log_{1+\varepsilon} n) = O(\frac{b^{i+1}}{\varepsilon^2}\log n)$ times with input string length $\frac{n}{b^i}$.

The time for running $\mathsf{ApproxLIS}$ and $\mathsf{ApproxLISBound}$ at the i-th recursive level is bounded by $O(b^i f(\frac{n}{b^{i-1}})) = o(\frac{1}{\log_b n} bf(n))$.

Since the depth of recursion is at most $\lceil \log_b n\rceil$, the total time used for running $\mathsf{ApproxLIS}$ and $\mathsf{ApproxLISBound}$ is bounded by $O(bf(n))$. Combining 2.13 and 2.15, we know the total running time is bounded by

$$O(bf(n)) = \Big(O(\frac{b^2}{\varepsilon^2}\log n)\Big)^{\lceil \log_b n\rceil - 2}\cdot (b^3\log n + \frac{b^2\log n}{\varepsilon \log b}) \tag{2.16}$$

□

As a direct result of Lemma 2.3.4, we have the following 2 lemmas.

Lemma 2.3.5. *Given a string $x \in \Sigma^n$, there is a deterministic algorithm that can output an increasing subsequence of length at least $(1-O(\frac{1}{\log\log n}))$ $\mathit{LIS}(x)$ with $O(\frac{\log^4 n}{\log\log n})$ bits of space in $O(n^{5+o(1)})$ time.*

Proof of Lemma 2.3.5. Let $b = \log n$ and $\varepsilon = \frac{1}{\log n}$. Then Theorem 2.3.5 is a direct result of Lemma 2.3.4. □

Lemma 2.3.6. *Given a string $x \in \Sigma^n$, for any constants $\delta \in (0, \frac{1}{2})$ such that $\frac{1}{\delta}$ is an integer, and $\varepsilon \in (0,1)$, there is a deterministic algorithm that can output an increasing subsequence of length at least $(1-\varepsilon)\,\mathsf{LIS}(x)$ with $\tilde{O}_{\varepsilon,\delta}(n^{\delta})$ bits of space in $\tilde{O}_{\varepsilon,\delta}(n^{2-\delta})$ time.*

Proof of Lemma 2.3.6. Let $b = n^{\delta}$ and pick ε' to be a constant sufficiently smaller than ε. Let $l = n$ and $\bar{s}$ be the output of $\mathsf{ApproxLIS}(x, b, \varepsilon')$. We run $\mathsf{LISSequence}(x, b, \varepsilon', l, \bar{s})$. Notice that when $\frac{1}{\delta}$ is an integer, we have $\lceil \log_b n \rceil = \lceil \frac{1}{\delta} \rceil = \frac{1}{\delta}$. Then Theorem 2.3.6 is a direct result of Lemma 2.3.4. □

2.4 Space Efficient Approximation for LCS

In this section, we describe our algorithm for approximating LCS with small space. Before introducing our algorithm, we introduce the following reduction from LCS to LIS.

Reducing LCS to LIS

Our space efficient algorithm for LCS is based on a reduction (algorithm 7) from LCS to LIS.

Algorithm 7: ReduceLCStoLIS

```
Data: Two strings $x \in \Sigma^n$ and $y \in \Sigma^m$.
1: initialize $z$ to be an empty string.
2: for $i = 1$ to $n$ do
3:     for $j = m$ to $1$ do
4:         if $x_i = y_j$ then
5:             add $j$ to the end of $z$.
6: return $z$.
```

Lemma 2.4.1. *Given two strings $x \in \Sigma^n$ and $y \in \Sigma^m$ as input to algorithm 7, let $z = \textsf{ReduceLCStoLIS}(x, y) \in [m]^*$ be the output, then the length of z is $O(mn)$ and $\textsf{LIS}(z) = \textsf{LCS}(x, y)$.*

Proof of Lemma 2.4.1. z can be viewed as the concatenation of n blocks such that $z = \hat{z}^1 \circ \hat{z}^2 \circ \cdots \circ \hat{z}^n$ ($\hat{z}^i$'s can be empty). For each i, the length of $\hat{z}^i$ is equal to the number of characters in y that are equal to x_i. The elements of $\hat{z}^i$ are the indices of characters in y that are equal to x_i and the indices in $\hat{z}^i$ are sorted in descending order. Since the length of $\hat{z}^i$ for each i is at most m, the length of z is at most mn.

Assuming $\mathsf{LIS}(z) = l$, we show $\mathsf{LCS}(x, y) \geq l$. By the assumption, there exists a subsequence of z with length l. We denote this subsequence by $t \in [m]^l$. Let $t = t_1 t_2 \cdots t_l$. Since $\hat{z}^i$'s are strictly descending, eash element in t is picked from a distinct block. We assume for each $i \in [l]$, t_i is picked from the block $\hat{z}^{t'_i}$. Then by the algorithm, we know $x_{t'_i} = y_{t_i}$. For $1 \leq i < j \leq l$, t_i appears before t_j. The block $\hat{z}^{t'_i}$ also appears before $\hat{z}^{t'_j}$. We have $1 \leq t'_1 < t'_2 < \cdots < t'_l \leq n$. Thus, $x_{t'_1} x_{t'_2} \cdots x_{t'_l}$ is a subsequence of x with length l and it is equal to $y_{t_1} y_{t_2} \cdots y_{t_l}$. Hence, $\mathsf{LCS}(x, y)$ is at least l.

On the other direction, assuming $\mathsf{LCS}(x, y) = l$, we show $\mathsf{LIS}(z) \geq l$. By the assumption, let $x' = x_{t'_1} x_{t'_2} \cdots x_{t'_l}$ be a subsequence of x and $y' = y_{t_1} y_{t_2} \cdots y_{t_l}$ be a subsequence of y such that $x' = y'$. Let $z' = \hat{z}^{t'_1} \circ \hat{z}^{t'_2} \circ \cdots \circ \hat{z}^{t'_l}$, which is a subsequence of z. For each $i \in [l]$, since $x_{t'_i} = y_{t_i}$, t_i appears in the block $\hat{z}^{t'_i}$. By $1 \leq t_1 < t_2 < \cdots < t_l \leq m$, we know $t = t_1 t_2 \cdots t_l$ is an increasing subsequence of z' and thus also an increasing subsequence of z. $\square$

Space Efficient Algorithm for LCS

Our goal is to compute the longest common subsequence between two strings x and y over alphabet Σ. We assume the input strings x and y both has length n and the alphabet size $|\Sigma|$ is polynomial in n. We call our space efficient algorithm for LCS ApproxLCS and give the pseudocode in algorithm 8.

The idea is to first reduce calculating $\mathsf{LCS}(x, y)$ to computing LIS with algorithm 7. We do not use ApproxLIS as a black box. Instead, we make slight modification to the approach to achieve better running time. We denote $z = \mathsf{ReduceLCStoLIS}(x, y)$. Although storing $z = \mathsf{ReduceLCStoLIS}(x, y)$ already takes $O(n^2 \log n)$ bits of space, we will show later that this is not required for our algorithm.

Similar to the case for LIS, we introduce a slightly modified version of ApproxLCS called ApproxLCSBound. It takes an additional input l. The modification are same: first, at line 9 of algorithm ApproxLCS, we always require k to be at most l. That is, we let $k = \min\{l, \max\{k, s + d\}\}$. Second, instead of output $\max S$, we output the whole set S and list Q (at the bottom level, we output the list maintained by PatienceSorting). We omit the pseudocode for ApproxLCSBound.

Lemma 2.4.2. *Given two strings $x, y \in \Sigma^n$, parameters $b \leq \sqrt{n}$ and $\varepsilon \in (0, 1)$, ApproxLCS(x, y, b, ε) computes a $(1 - 3\log_b(n)\varepsilon)$ approximation of LCS(x, y) with $O(\frac{b \log^2 n}{\varepsilon \log b})$ bits of space in $\left(O(\frac{b^2 \log n}{\varepsilon^2})\right)^{\lceil \log_b n \rceil - 1} \cdot bn \log n$ time.*

Proof. When the input string x is of length at most b, let $z = \mathsf{ReduceLCStoLIS}(x, y)$. Since z is consists of at most b parts, all in decreasing order. $\mathsf{LIS}(z) \leq b$. We can compute $\mathsf{LIS}(z)$ using PatienceSorting with $O(b \log n)$ space and $O(bn \log n)$ time. We do not need to store z. This is because PatienceSorting only need to scan z from left to right once. We can do this by scanning y from right to left b times. The total time is still $O(bn \log n)$.

Otherwise, let $z = \mathsf{ReduceLCStoLIS}(x, y)$. We use the same notation as in the proof of Lemma 2.4.1 such that $z \in [n]^{O(n^2)}$ can be viewed as the concatenation of n blocks. That is, $z = \hat{z}^1 \circ \hat{z}^2 \circ \cdots \circ \hat{z}^n$, where $\hat{z}^i$ consists of indices of characters in y that are equal to x^i, arranged in descending order.

Algorithm 8: ApproxLCS (ApproxLCSBound): space efficient approximation of LCS

```
Data: Two strings $x, y \in \Sigma^*$, parameters $b$ and $\varepsilon$. An additional parameter $l$
      for ApproxLCSBound
 1: if $|x| \leq b$ then
 2:     let $z =$ ReduceLCStoLIS$(x, y)$ compute LIS$(z)$ exactly with $O(b \log n)$ space in
        $O(|x||y|) = O(bn \log n)$ time with PatienceSorting. (for ApproxLCSBound, we
        only consider LIS of length at most $l$)
 3: divide $x$ evenly into $b$ blocks such that $x = x^1 \circ x^2 \circ \cdots \circ x^b$.
 4: initialize $S = \{0\}$ and $Q[0] = -\infty$ .
 5: for $i = 1$ to $b$ do
 6:     $k = 0$.
 7:     for all $s \in S$ do
 8:         $d =$ ApproxLCS$(x^i, y^*(Q[s]), b, \varepsilon)$ ▷ by $y^*(Q[s])$, we mean the string we get by
            replacing the first $Q[s]$ elements of $y$ with a special symbol $*$ that does not
            appear in $x$ .
 9:         $k = \max\{k, s + d\}$ (for ApproxLCSBound, we let $k = \min\{l, \max\{k, s + d\}\}$).
10:     if $k \leq b/\varepsilon$, let $S' = \{0, 1, 2, \ldots, k\}$, otherwise let $S' = \{0, \frac{\varepsilon}{b}k, 2\frac{\varepsilon}{b}k, \ldots, k\}$ ▷
        evenly pick $\frac{b}{\varepsilon} + 1$ integers from $0$ to $k$ (including $0$ and $k$) .
11:     $Q'[s] \leftarrow \infty$ for all $s \in S'$ except $Q'[0] = -\infty$.
12:     for all $s \in S$ do
13:         for all $l = 1, 1 + \varepsilon, (1 + \varepsilon)^2, \ldots, k - s$ do
14:             $\tilde{S}, \tilde{Q} \leftarrow$ ApproxLCSBound$(x^i, y^*(Q[s]), b, \varepsilon, l)$ ;
                for each $s' \in S'$ such that $s \leq s' \leq s + l$, let $\tilde{s}$ be the smallest element in $\tilde{S}$
                that is larger than $s' - s$ and set $Q'[s'] = \min\{\tilde{Q}[\tilde{s}], Q'[s']\}$.
15:     $S \leftarrow S'$, $Q \leftarrow Q'$.
16: return $\max\{s \in S\}$. (for ApproxLCSBound, we return $S$ and $Q$)
```

The recursion stops when the first input string has length no larger than b. Also, every time we enter next level, the length of input string x is decreased by a factor of b. The depth of recursion is at most $\lceil \log_b n \rceil$.

Our algorithm ApproxLCS is essentially computing the length of LIS of z. However, unlike the algorithm ApproxLIS, instead of partition z equally into b blocks, we partition z according to x. That is, we first evenly divide x into b blocks such that $x = x^1 \circ x^2 \circ \cdots \circ x^b$. z is then naturally divided into b blocks $z = z^1 \circ z^2 \circ \cdots \circ z^b$ (note that z^i is not the same as $\hat{z}^i$), where $z^i = \mathsf{ReduceLCStoLIS}(x^i, y)$. Thus, $\mathsf{ApproxLCS}(x^i, y, b, \varepsilon)$ computes a good approximation of $\mathsf{LIS}(z^i)$.

In our algorithm, we use the notation $y^*(Q[s])$ to denote the string we get by replacing the first $Q[s]$ characters of y with a special symbol $*$ that does not appear in x. Thus, $\mathsf{ReduceLCStoLIS}(x^i, y^*(Q[s]))$ is the subsequence of $\mathsf{ReduceLCStoLIS}(x^i, y)$ with only elements larger than $Q[s]$. By running $\mathsf{ApproxLCS}(x^i, y^*(Q[s]), b, \varepsilon)$, we are computing a good approximation of the length of LIS of z^i with first element larger than $Q[s]$.

The proof of correctness of algorithm ApproxLCS then follows directly from that of algorithm ApproxLIS.

Notice that it is not required to stored string z at any level of the algorithm. We divide z according to the corresponding position in x and we only need to query z at the last level.

We now turn to space and time complexity. The analysis is similar to that of algorithm ApproxLIS except

ApproxLCS is a recursive algorithm. We start by analyse the recursion tree. Notice that except at the bottom level, ApproxLCS needs to call itself $O(b|S|) = O(\frac{b^2}{\varepsilon})$ times and ApproxLCSBound $O(b|S| \log_{1+\varepsilon} n) = O(\frac{b^2 \log n}{\varepsilon^2})$ (since $|S|$ is at most $\frac{b}{\varepsilon}$) times. Thus, the degree of the recursion tree is $O(\frac{b^2 \log n}{\varepsilon^2})$. Also, as we have shown, the depth of recursion is at most $\log_b n$.

The computation of ApproxLCS has the same order as doing depth-first search on the recursion tree. We only need to remember the information in one node at each level.

For the inner nodes of the recursion, ApproxLCS maintains a set S and a list Q, both of size $\frac{b}{\varepsilon}$. Since each elements in S and Q takes $O(\log n)$ bits. The space needed for an inner node is $O(\frac{b}{\varepsilon}\log n)$. For the leaf node, we compute $\mathsf{LCS}(x, y)$ exactly with $O(b \log n)$ space. Thus, the total space required for ApproxLCS is $O(d(\frac{b}{\varepsilon}\log n))$ where d is the depth of recursion. Since $d \le \lceil \log_b n \rceil = \lceil \frac{\log n}{\log b} \rceil$, we know running ApproxLCS takes $O(\frac{b \log^2 n}{\varepsilon \log b})$.

For the time complexity, we denote the used within i-th level by T_i (excluding the time used for running itself or ApproxLCSBound) and the total running time by T. We have $T = \sum_{i=1}^{d} T_i$.

For the leaf nodes, we run exact algorithm with $O(bn \log n)$ time. Since the recursion tree has degree $O(\frac{b^2 \log n}{\varepsilon^2})$ and depth $\log_b n$, the number of nodes at d-th level (leaf nodes) is bounded by $(O(\frac{b^2 \log n}{\varepsilon^2}))^{\log_b n - 1}$. We have

$$T_d = \Big(O(\frac{b^2 \log n}{\varepsilon^2})\Big)^{\lceil \log_b n \rceil - 1} \cdot bn \log n. \tag{2.17}$$

For the inner nodes, the time is dominated by the operations at line 14 of algorithm 8. Since the size of S and $\tilde{S}$ are both at most $\frac{b}{\varepsilon}$ and we try at most $\log_{1+\varepsilon} n$ different l, it takes $O(b|S|^2 \log_{1+\varepsilon} n) = O(\frac{b^3}{\varepsilon^3} \log n)$ time. Also, the number of nodes at i-th level is bounded by $(O(\frac{b^2 \log n}{\varepsilon^2}))^{i-1}$. We have

$$T_i = \Big(O(\frac{b^2 \log n}{\varepsilon^2})\Big)^{i-1} \cdot \frac{b^3}{\varepsilon^3} \log n. \tag{2.18}$$

Thus

$$\begin{aligned}\sum_{i=1}^{d-1} T_i &\le (d-1)\cdot T_{d-1} \\ &\le d\cdot\Big(O(\frac{b^2\log n}{\varepsilon^2})\Big)^{d-2}\cdot\frac{b^3}{\varepsilon^3}\log n \\ &= \Big(O(\frac{b^2\log n}{\varepsilon^2})\Big)^{d-1}\cdot\frac{b\log n}{\varepsilon\log b} \\ &= \Big(O(\frac{b^2\log n}{\varepsilon^2})\Big)^{\lceil\log_b n\rceil-1}\cdot\frac{b\log n}{\varepsilon\log b}\end{aligned} \tag{2.19}$$

Compare 2.17 and 2.19, we know that the total running time is dominated by T_d. We have

$$T = \Big(O(\frac{b^2\log n}{\varepsilon^2})\Big)^{\lceil\log_b n\rceil-1}\cdot bn\log n.$$

□

Lemma 2.4.3. *Given two strings string $x, y \in \Sigma^n$, there is a deterministic algorithm that computes a $1-O(\frac{1}{\log\log n})$ approximation of $\mathsf{LCS}(x,y)$ with $O(\frac{\log^4 n}{\log\log n})$ bits of space in $O(n^{6+o(1)})$ time.*

Proof of Lemma 2.4.3. Let $b=\log n$ and $\varepsilon=\frac{1}{\log n}$. Then Theorem 2.4.3 is a direct result of Lemma 2.4.2. □

Lemma 2.4.4. *Given two strings $x, y \in \Sigma^n$, for any constant $\delta\in(0,\frac{1}{2})$ and $\varepsilon\in(0,1)$, there is a deterministic algorithm that computes a $1-\varepsilon$ approximation of $\mathsf{LCS}(x,y)$ with $\tilde{O}_{\varepsilon,\delta}(n^\delta)$ bits of space in $\tilde{O}_{\varepsilon,\delta}(n^{3-\delta})$ time.*

Proof of Lemma 2.4.4. Let $b=n^\delta$ and pick ε' to be a constant sufficiently smaller than ε. We run $\mathsf{ApproxLCS}(x,y,b,\varepsilon')$. Then Theorem 2.4.4 is a direct result of Lemma 2.4.2. □

Theorem 2.2 is a direct result of Lemma 2.4.3 and Lemma 2.4.4.

Output the Subsequence We now show how to output the common subsequence we have detected with small space. The idea is similar to our approach for outputting the approximate longest increasing subsequence.

Similarly, let b be a parameter we will pick later. For the base case, we use the linear space alogrithm from [117] that output a LCS of x and y with $O(\min(n, m) \log n)$ space (we assume alphabet size is polynomial in n). Thus, one of the string has length no larger than b, we can output the longest common subsequence with $O(b \log n)$ space.

We call our algorithm for outputing the sequence LCSSequence. The pseudocode can be found in algorithm 9.

Algorithm 9: LCSSequence: output the subsequence detected by ApproxLCS

```
Data: Two strings $x, y \in \Sigma^*$ and parameters $b$ and $\varepsilon$, $l$, $\bar{s}$.
 1: if $|x| \le b$ or $|y| \le b$ then
 2:     Output the longest common subsequence of $x$ and $y$ with $O(b \log n)$ bits of
        space and $O(bn)$ time.
 3: divide $x$ evenly into $b$ blocks $x^1 \circ x^2 \circ \cdots \circ x^b$.
 4: compute $S_b$ and $Q_b$ by running ApproxLCSBound$(x, y, b, \varepsilon, l)$.      ▷ $S_i$ and $Q_i$ are
    the set $S$ and list $Q$ after $i$-th outer loop of ApproxLCSBound .
 5: set $B$ to be a list with $B[0] = -\infty$
 6: $B[b] = Q_b[\bar{s}]$ where $s_b = \bar{s}$.                       ▷ we guarantee that $\bar{s} \in S_b$
 7: for $i = b - 1$ to $1$ do
 8:     release the space used for storing $S_{i+1}$, and $Q_{i+1}$.
 9:     compute $S_i$, $Q_i$ by running ApproxLCSBound$(x, y, b, \varepsilon, l)$ .
10:     for all $s \in S_i$ such that $s \le s_{i+1}$ do
11:         let $z$ be the subsequence of $x^i$ by only considering the elements larger than
            $Q[s]$.
12:         for all $l = 1, 1 + \varepsilon, (1 + \varepsilon)^2, \dots, k - s$ do
13:             $\tilde{S}, \tilde{Q} \leftarrow$ ApproxLCSBound$(x^i, y^*(Q[s]), \varepsilon, l)$.     ▷ by $y^*(Q[s])$, we mean the
                string we get by replacing the first $Q[s]$ elements of $y$ with a special symbol
                $*$ that does not appear in $x$ .
14:             if there is an $\tilde{s} \in \tilde{S}$ such that $\tilde{s} + s \ge s_{i+1}$ and $B[i + 1] = \tilde{Q}[\tilde{s}]$, we set
                $B[i] = Q_i[s]$, $l_i = \tilde{l}$, $s_i = s$, $\bar{s}_i = \tilde{s}$ and continue the loop at line 7
15: for $i = 1$ to $b$ do
16:     LCSSequence$(x^i, y[B[i - 1] + 1 : n], b, \varepsilon, l_i, \bar{s}_i)$.
```

Lemma 2.4.5. *Given two strings $x, y \in \Sigma^n$ and two parameters $b \le \sqrt{n}$ and*

$\varepsilon \in (0,1)$, setting $l = n$ and let $\bar{s}$ be the output of $\textsf{ApproxLCS}(x, y, b, \varepsilon, n)$, then $\textsf{LCSSequence}(x, y, b, \varepsilon, l, \bar{s})$ outputs an increasing subsequence of x with length at least $(1 - 3\log_b(n)\varepsilon)\,\textsf{LCS}(x, y)$ with $O(\frac{b\log^2 n}{\varepsilon \log b})$ bits of space in $\left(O(\frac{b^2 \log n}{\varepsilon^2})\right)^{\lceil \log_b n \rceil - 1} \cdot b^2 n \log n$ time.

Proof of Lemma 2.4.5. Algorithm LCSSequence is a modified version of LISSequence. One difference is that, when the input string x has length at most b, we output the longest common subsequence of x and y with $O(b \log n)$ bits of space and $O(bn)$ time. This can be achieved by using the linear space algorithm from [117].

Our algorithm for approximating LCS is base on the reduction from LCS to LIS. Let $z = \textsf{ReduceLCStoLIS}(x, y)$. Instead of output an increasing subsequence of z, we need to output the corresponding common subsequence of x and y. Similarly to our analysis of algoirhm LISSequence. Let ρ be the longest increasing subsequence we have detected in z. We can divide ρ into b blocks such that ρ^i lies in $z^i = \textsf{ReduceLCStoLIS}(x^i, y)$. The list B here serves the same purpose as in the algorithm LISSequence. Namely, $B[i]$ is equal to the last element of ρ^i. By the correctness of algorithm LCSSequence, we are guaranteed to output a common subsequence of x and y with length at least $(1 - 3\log_b(n)\varepsilon)\,\textsf{LCS}(x, y)$.

For the time and space complexity, the analysis is also the same as that of LISSequence. The space is dominated by the space used for running ApproxLCS. The time complexity is $O(bf(n))$ where $f(n)$ is the running time of ApproxLCS when the first input string has length n.

□

Lemma 2.4.6. *Given two strings $x, y \in \Sigma^n$, there is a deterministic algorithm that can output an common subsequence of length at least $(1 - O(\frac{1}{\log\log n}))\,\textsf{LCS}(x, y)$ with $O(\frac{\log^4 n}{\log\log n})$ bits of space in $O(n^{6+o(1)})$ time.*

Proof of Lemma 2.4.6. Let $b = \log n$ and $\varepsilon = \frac{1}{\log n}$. Then Theorem 2.4.6 is a direct result of Lemma 2.4.5. □

Lemma 2.4.7. *Given two strings $x, y \in \Sigma^n$, for any constant $\delta \in (0, \frac{1}{2})$ such that $\frac{1}{\delta}$ is an integer, and $\varepsilon \in (0,1)$, there is a deterministic algorithm that can output an common subsequence of length at least $(1-\varepsilon)\,\mathsf{LCS}(x,y)$ with $\tilde{O}_{\varepsilon,\delta}(n^\delta)$ bits of space in $\tilde{O}_{\varepsilon,\delta}(n^3)$ time.*

Proof of Lemma 2.4.7. Let $b = n^\delta$ and pick ε' to be a constant sufficiently smaller than ε. We run $\mathsf{LCSSequence}(x, y, b, \varepsilon')$. Notice that when $\frac{1}{\delta}$ is an integer, we have $\lceil \log_b n \rceil = \lceil \frac{1}{\delta} \rceil = \frac{1}{\delta}$. Then Theorem 2.4.7 is a direct result of Lemma 2.4.5. □

2.5 Open Problems

Our work leaves many interesting open problems, and we list some of them below.

1. Can we achieve better space complexity or better time complexity, or both? For example, is it possible to further reduce the space complexity to even logarithmic while still maintaining polynomial running time? Or can we maintain poly-logarithmic space, but also achieve quadratic or even sub-quadratic time complexity? What kind of approximations can we achieve in these cases? For example, can we keep the approximation factor to be $1+\varepsilon$ or $1-\varepsilon$, or a constant? We believe it requires new ideas to answer these questions. We remark that in this direction, a recent work [118] provides randomized algorithms which can give a constant factor approximation to ED in both slightly sub-linear space and slightly sub-quadratic time. It remains to see if one can do better or design a similar deterministic algorithm.

2. So far all our algorithms are deterministic. How does randomness help here? Can we design randomized algorithms that achieve $1+\varepsilon$ or $1-\varepsilon$ approximation, but with better space complexity?

3. Finally, is there a good reason for the lack of progress on computing edit distance and longest common subsequence *exactly* using polynomial time and strongly sub linear space? In other words, it would be nice if one can provide justification like the SETH-hardness of computing edit distance and longest common subsequence exactly in truly sub-quadratic time. We note that a recent work of Yamakami [119] proposes a so called *Linear Space Hypothesis*, which conjectures that some NL-complete problems cannot be solved simultaneously in polynomial time and strongly sub linear space. Thus it would be nice to show reductions from these problems to edit distance and longest common subsequence. We note that here we need a reduction that simultaneously uses small space and polynomial time.

Chapter 3

Asymmetric Streaming Model: Algorithms

3.1 Introduction

In this chapter, we present several algorithms for approximating ED and LCS in the asymmetric streaming model.

3.1.1 Main Results

The first result is inspired by our space efficient approximation algorithms. For any small constant $\varepsilon > 0$, we can get $1 \pm \varepsilon$ approximation of ED and LCS with $\tilde{O}(\frac{\sqrt{n}}{\varepsilon})$ space in polynomial time.

Theorem 3.1. *In the asymmetric streaming model, for any small constant $\varepsilon > 0$, there are one-pass deterministic algorithms that out puts a $(1+\varepsilon)$ approximation of LCS or a $1-\varepsilon$ approximation of ED using $\tilde{O}\left(\frac{\sqrt{n}}{\varepsilon}\right)$ bits of space and in polynomial time.*

We note that in [39], the authors studied the approximation algorithm for insertion deletion distance in the asymmetric streaming model where the insertion deletion distance ($\widetilde{\mathsf{ED}}$) is defined as the minimum number of insertions and deletions required to transform x to y. [39] gives an algorithm that outputs a $(1+\varepsilon)$ approximation of

$\widetilde{\mathsf{ED}}(x, y)$ for any constant $\varepsilon > 0$ with $\tilde{O}_\varepsilon(\sqrt{n})$ space. The algorithm also works for edit distance through a simple reduction. We provide a proof in Section 3.2.1

Also note that for LCS over large alphabet, this upper bound matches the lower bounds implied by [42, 43] (More discussion in Chapter 4).

The second result is for ED. The key idea of the algorithm is first described in [18]. It gives a $O(2^{1/\delta})$-approximation algorithm with $\tilde{O}(n^\delta)$ space for any constant $\delta \in (0, 1)$. In [41], with some new observations, the space complexity is further reduced from $\tilde{O}(\frac{n^\delta}{\delta})$ to $O\Big(\frac{d^\delta}{\delta} \text{ polylog}(n)\Big)$ where $d = \mathsf{ED}(x, y)$. Specifically, we have the following theorem.

Theorem 3.2. *Assume ED$(x, y) = d$, in the asymmetric streaming model, there are one-pass deterministic algorithms in polynomial time with the following parameters:*

1. *A $(3 + \varepsilon)$-approximation of ED(x, y) using $O(\sqrt{d} \text{ polylog}(n))$ space.*
2. *For any constant $\delta \in (0, 1/2)$, a $2^{O(\frac{1}{\delta})}$-approximation of ED(x, y) using $O\Big(\frac{d^\delta}{\delta} \text{ polylog}(n)\Big)$ space.*

The last result is about LCS over small alphabet. Note that our Theorem 4.3 does not give anything useful if $|\Sigma|$ is small and ε is large (e.g., both are constants). Thus a natural question is whether one can get better bounds. In particular, is the dependence on $1/\varepsilon$ linear as in our theorem, or is there a threshold beyond which the space jumps to say for example $\Omega(n)$? We note that there is a trivial one pass, $O(\log n)$ space algorithm even in the standard streaming model that gives a $|\Sigma|$ approximation of LCS (or $1/|\Sigma|$ approximation in standard notation), and no better approximation using sublinear space is known even in the asymmetric streaming model. Thus one may wonder whether this is the threshold. We show that this is not the case, by giving a one pass algorithm in the asymmetric streaming model over the binary alphabet that achieves a $2 - \varepsilon$ approximation of LCS (or $1/2 + \varepsilon$ approximation in standard notation), using space n^δ for any constant $\delta > 0$.

Theorem 3.3. *For any constant $\delta \in (0, 1/2)$, there exists a constant $\varepsilon > 0$ and a one-pass deterministic algorithm that outputs a $2 - \varepsilon$ approximation of $\mathsf{LCS}(x, y)$ for any two strings $x, y \in \{0, 1\}^n$, with $\tilde{O}(n^\delta/\delta)$ space and polynomial time in the asymmetric streaming model.*

3.1.2 Overview of Techniques

Edit Distance The $(1+\varepsilon)$ approximation algorithm for ED using $\tilde{O}\left(\frac{\sqrt{n}}{\varepsilon}\right)$ space is a special case of our space efficient approximation algorithm presented in Chapter 2. To see this, notice that the block decomposition of the string x can be viewed as a tree, and for a fixed sequence of Δ in each level of recursion, the algorithm we discussed in Chapter 2 is essentially doing a depth first search on the tree, which implies a streaming computation on x. However, the requirement to try all possible Δ and all candidate intervals may ruin this property since we need to traverse the tree multiple times. To avoid this, our idea is to simultaneously keep track of all possible Δ and candidate intervals in the depth first search tree on x. We stop the recursion and do exact computation whenever each block of x is no larger than $\sqrt{n}$. By doing so, we can still bound the space usage by $\tilde{O}(\sqrt{n})$.

For the $2^{O(\frac{1}{\delta})}$-approximation using $O\left(\frac{d^\delta}{\delta} \text{ polylog}(n)\right)$ space, the key idea is to use triangle inequality. Given a constant δ, the algorithm first divides x evenly into $b = n^\delta$ blocks. Then for each block x^i of x, the algorithm recursively finds an α-approximation of the closest substring to x^i in y. That is, the algorithm finds a substring $y[l_i : r_i]$ and a value d_i such that for any substring $y[l : r]$ of y, $\mathsf{ED}(x^i, y[l_i : r_i]) \le d_i \le \alpha\, \mathsf{ED}(x^i, y[l : r])$. Let $\tilde{y}$ be the concatenation of $y[l_i : r_i]$ from $i = 1$ to b. Then using triangle inequality, we can show that $\mathsf{ED}(y, \tilde{y}) + \sum_{i=1}^{b} d_i$ is a $2\alpha + 1$ approximation of $\mathsf{ED}(x, y)$. The $\tilde{O}(n^\delta)$ space is achieved by recursively applying this idea, which results in a $O(2^{1/\delta})$ approximation. These ideas are first presented in [18].

To further reduce the space complexity, our key observation is that, instead of dividing x into blocks of equal length, we can divide it according to the positions of the edit operations that transform x to y. More specifically, assume we are given a value k with $\mathsf{ED}(x, y) \leq k \leq c\,\mathsf{ED}(x, y)$ for some constant c, we show how to design an approximation algorithm using space $\tilde{O}(\sqrt{k})$. Towards this, we can divide x and y each into $\sqrt{k}$ blocks $x = x^1 \circ \cdots \circ x^{\sqrt{k}}$ and $y = y^1 \circ \cdots \circ y^{\sqrt{k}}$ such that $\mathsf{ED}(x^i, y^i) \leq \frac{\mathsf{ED}(x,y)}{\sqrt{k}} \leq \sqrt{k}$ for any $i \in [\sqrt{k}]$. However, such a partition of x and y is not known to us. Instead, we start from the first position of x and find the largest index l_1 such that $\mathsf{ED}(x[1 : l_1], y[p_1, q_1]) \leq \sqrt{k}$ for some substring $y[p_1 : q_1]$ of y. To do this, we start with $l = \sqrt{k}$ and try all substrings of y with length in $[l - \sqrt{k}, l + \sqrt{k}]$. If there is some substring of y within edit distance $\sqrt{k}$ to $x[1 : l]$, we set $l_1 = l$ and store all the edit operations that transform $y[p_1 : q_1]$ to $x[1 : l_1]$ where $y[p_1 : q_1]$ is the substring closest to $x[1 : l_1]$ in edit distance. We continue doing this with $l = l + 1$ until we can not find a substring of y within edit distance $\sqrt{k}$ to $x[1 : l]$.

One problem here is that l can be much larger than $\sqrt{k}$ and we cannot store $x[1 : l]$ with $\tilde{O}(\sqrt{k})$ space. However, since we have stored some substring $y[p_1 : q_1]$ (we only need to store the two indices p_1, q_1) and the at most $\sqrt{k}$ edit operations that transform $y[p_1 : q_1]$ to $x[1 : l - 1]$, we can still query every bit of $x[1 : l]$ using $\tilde{O}(\sqrt{k})$ space.

After we find the largest possible index l_1, we store l_1, (p_1, q_1) and $d_1 = \mathsf{ED}(x[1 : l_1], y[p_1 : q_1])$. We then start from the $(l_1 + 1)$-th position of x and do the same thing again to find the largest l_2 such that there is a substring of y within edit distance $\sqrt{k}$ to $x[l_1 + 1 : l_1 + l_2]$. We continue doing this until we have processed the entire string x. Assume this gives us T pairs of indices (p_i, q_i) and integers l_i, d_i from $i = 1$ to T, we can use $O(T \log n)$ space to store them. We show by induction that $x^1 \circ \cdots \circ x^i$ is a substring of $x[1 : \sum_{j=1}^{i} l_j]$ for $i \in [T - 1]$. Recall that $x = x^1 \circ \cdots \circ x^{\sqrt{k}}$ and each $l_i > 0, i \in [T - 1]$. Thus, the process must end within $\sqrt{k}$ steps and we have $T \leq \sqrt{k}$. Then, let $\tilde{y}$ be the concatenation of $y[p_i : q_i]$ from $i = 1$ to T. We can show $\mathsf{ED}(y, \tilde{y}) + \sum_{i=1}^{T} d_i$ is a 3 approximation of $\mathsf{ED}(x, y)$. For any small constant $\varepsilon > 0$, we can compute a $1 + \varepsilon$ approximation of $\mathsf{ED}(y, \tilde{y})$ with polylog(n) space using the algorithm in [17]. This gives us a $3 + \varepsilon$ approximation algorithm with $O(\sqrt{\mathsf{ED}(x, y)}\ \text{polylog}(n))$ space.

We then use recursion to further reduce the space. Let δ be a small constant and a value $k = \Theta(\mathsf{ED}(x, y))$ be given as before. There is a way to partition x and y each into k^δ blocks such that $\mathsf{ED}(x^i, y^i) \leq \frac{\mathsf{ED}(x,y)}{k^\delta} \leq k^{1-\delta}$. Now similarly, we want to find the largest index l^0 such that there is a substring of y within edit distance $k^{1-\delta}$ to $x[1 : l^0]$. However naively this would require $\Theta(k^{1-\delta})$ space to compute the edit distance. Thus again we turn to approximation.

We introduce a recursive algorithm called $\mathsf{FindLongestSubstring}$. It takes two additional parameters as inputs: an integer u and a parameter s for the amount of space we can use. It outputs a three tuple: an index l, a pair of indices (p, q) and an integer d. Let l^0 be the largest index such that there is a substring of y within edit distance u to $x[1 : l^0]$.

We show the following two properties of $\mathsf{FindLongestSubstring}$: (1) $l \geq l^0$, and (2) for any substring $y[p^* : q^*]$, $\mathsf{ED}(x[1 : l], y[p : q]) \leq d \leq c(u, s)\, \mathsf{ED}(x[1 : l], y[p^* : q^*])$. Here, $c(u, s)$ is a function of (u, s) that measures the approximation factor. If $u \leq s$, $\mathsf{FindLongestSubstring}$ outputs $l = l^0$ and the substring of y that is closest to $x[1 : l]$ using $O(s \log n)$ space by doing exact computation. In this case we set $c(u, s) = 1$. Otherwise, it calls $\mathsf{FindLongestSubstring}$ itself up to s times with parameters u/s and s. This gives us $T \leq s$ outputs $\{l_i, (p_i, q_i), d_i\}$ for $i \in [T]$. Let $\tilde{y}$ be the concatenation of $y[p_i : q_i]$ for $i = 1$ to T. We find the pair of indices (p, q) such that $y[p : q]$ is the substring that minimizes $\mathsf{ED}(\tilde{y}, y[p : q])$. We output $l = \sum_{j=1}^{T} l_j$, (p, q), and $d = \mathsf{ED}(\tilde{y}, y[p : q]) + \sum_{i=1}^{T} d_i$. We then use induction to show property (1) and (2) hold for these outputs, where $c(u, s) = 2(c(u/s, s) + 1)$ if $u > s$ and $c(u, s) = 1$ if $u \leq s$. Thus we have $c(u, s) = 2^{O(\log_s u)}$.

This gives an $O(k^\delta/\delta \text{ polylog}(n))$ space algorithm as follows. We run algorithm $\mathsf{FindLongestSubstring}$ with $u = k^{1-\delta}$ and $s = k^\delta$ to find T tuples: $\{l_i, (p_i, q_i), d_i\}$. Again, let $\tilde{y}$ be the concatenation of $y[p_i : q_i]$ from $i = 1$ to T. Similar to the $O(\sqrt{k} \text{ polylog}(n))$ space algorithm, we can show $T \leq k^\delta$ and $\mathsf{ED}(y, \tilde{y}) + \sum_{i=1}^{T} d_i$ is a $2c(k^{1-\delta}, k^\delta) + 1 = 2^{O(1/\delta)}$ approximation of $\mathsf{ED}(x, y)$. Since the depth of recursion is at most $1/\delta$ and each level of recursion needs $O(k^\delta \text{ polylog}(n))$ space, $\mathsf{FindLongestSubstring}$ uses $O(k^\delta/\delta \text{ polylog}(n))$ space.

The two algorithms above both require a given value k. To remove this constraint, our observation is that the two previous algorithms actually only need the number k to satisfy the following relaxed condition: there is a partition of x into k^δ blocks such that for each block x^i, there is a substring of y within edit distance $k^{1-\delta}$ to x^i. Thus, when such a k is not given, we can do the following. We first set k to be a large constant k_0. While the algorithm reads x from left to right, let T' be the number of $\{l_i, (p_i, q_i), d_i\}$ we have stored so far. Each time we run $\mathsf{FindLongestSubstring}$ at this level, we increase T' by 1. If the current k satisfies the relaxed condition, then by a similar argument as before T' should never exceed k^δ. Thus whenever $T' = k^\delta$, we increase k by a $2^{1/\delta}$ factor. Assume that k is updated m times in total and after the i-th update, k becomes k_i. We show that $k_m = O(\mathsf{ED}(x, y))$ (but k_m may be much smaller than $\mathsf{ED}(x, y)$). To see this, suppose $k_j > 2^{1/\delta}\,\mathsf{ED}(x, y)$ for some $j \leq m$. Let t_j be the position of x where k_{j-1} is updated to k_j. We know it is possible to divide $x[t_j : n]$ into $\mathsf{ED}(x, y)^\delta$ blocks such that for each part, there is a substring of y within edit distance $\mathsf{ED}(x, y)^{1-\delta} \leq k_j^{1-\delta}$ to it. By property (1) and a similar argument as before, we will run $\mathsf{FindLongestSubstring}$ at most $\mathsf{ED}(x, y)^\delta$ times until we reach the end of x. Since $k_j^\delta - k_{j-1}^\delta > \mathsf{ED}(x, y)^\delta$, T' must be always smaller than k_j^δ and hence k_j will not be updated. Therefore we must have $j = m$. This shows $k_{m-1} \leq 2^{1/\delta}\,\mathsf{ED}(x, y)$ and $k_m \leq 2^{2/\delta}\,\mathsf{ED}(x, y)$. Running $\mathsf{FindLongestSubstring}$ with $k \leq k_m$ takes $O(k_m^\delta/\delta \text{ polylog}(n)) = O(\mathsf{ED}(x, y)^\delta/\delta \text{ polylog}(n))$ space and the number of intermediate results ((p_i, q_i) and d_i's) is $O(k_m^\delta) = O(\mathsf{ED}(x,y)^\delta)$. This gives us a $2^{O(1/\delta)}$ approximation algorithm with space complexity $O(\mathsf{ED}(x,y)^\delta/\delta \text{ polylog}(n))$.

Longest Common Subsequence The algorithm that outputs $(1-\varepsilon)$-approximation of LCS using $\tilde{O}\left(\frac{\sqrt{n}}{\varepsilon}\right)$ space is a special case of the algorithm $\mathsf{ApproxLCS}$ (Algorithm 8). Specifically, we set $b = \sqrt{n}$, and the depth of recursion is 2. When processing block x^i, we can store the whole block in the memory with $\tilde{O}(\sqrt{n})$ space. Since we only process x^i from $i = 1$ to b once, we only need to read x from left to right once. Thus, our algorithm is a streaming algorithm that queries x in one pass using $O(\frac{\sqrt{n}}{\varepsilon}\log n)$ bits of space.

The second result (Theorem 3.3) is achieved by showing that the reduction from LCS to ED discovered in [11] can work in the asymmetric streaming model with a slight modification. Combined with our algorithm for ED, this gives a n^δ space algorithm for LCS that achieves a $1/2 + \varepsilon$ approximation for binary strings. We stress that the original reduction in [11] is not in the asymmetric streaming model and hence our result does not follow directly from previous works.

3.2 Edit Distance

In this section, we give details of our space efficient algorithms for approximating edit distance in the asymmetric streaming setting.

3.2.1 $(1+\varepsilon)$-Approximation using $\tilde{O}\left(\frac{\sqrt{n}}{\varepsilon}\right)$ Space.

Proof of Theorem 3.1 (The edit distance part). The algorithm is a slight modification of $\mathsf{ApproxED}$ (Algorithm 4). In that algorithm, we take $b = \sqrt{n}$ and ε' to be a constant sufficiently smaller than the given positive constant ε.

The idea is to run the for-loop starting from line 7 of Algorithm 4 in parallel. This creates $O(\log_{1+\varepsilon} n) = O(\frac{\log n}{\varepsilon})$ parallel instances. Finally, we output the smallest edit distance find by these instances. This does not change the result of $\mathsf{ApproxED}(x, y, b, \varepsilon')$.

We only need to show each instance takes $\tilde{O}(\sqrt{n})$ bits of space and reads x from left to right once. Notice that the computation in each instance is the same as running $\mathsf{ApproxED}$ except that we only try one Δ instead of all $\Delta = (1+\varepsilon)^j$ for $j \in [\lceil \log_{1+\varepsilon} n \rceil]$. Thus, it has the same space complexity as $\mathsf{ApproxED}$ and can be computed with $\tilde{O}(\sqrt{n})$ bits of space. Running them in parallel increase the aggregated space by a factor of $O(\log_{1+\varepsilon} n)$

The depth of recursion is 2 when $b = \sqrt{n}$. We only need to query x in the second level and we query each block of x one by one. That is, we only query block x^{i+1} after we have finished the computation on input x^i. Thus, when we need to query block x^i, we can store the whole block with $O(\sqrt{n}\log n)$ bits of space. After we have finished the computation on block x^i, we can release the space and scan the next $\sqrt{n}$ elements of x. This only adds another $O(\sqrt{n}\log n)$ bits to the aggregated space and we only need to scan x from left to right once. □

We also provide a proof that the algorithms presented in [39] for insertion deletion distance can also work for edit distance through a simple reduction. Specifically, Saks and Seshadri [39] studied the approximation algorithm for insertion deletion distance in the asymmetric streaming model. Here, the insertion deletion distance ($\widetilde{\mathsf{ED}}$) is defined as the minimum number of insertions and deletions required to transform x to y. Thus, assuming $|x| = |y| = n$, $\widetilde{\mathsf{ED}}(x, y) = 2(n - \mathsf{LCS}(x, y))$. [39] gives an algorithm that outputs a $(1 + \varepsilon)$ approximation of $\widetilde{\mathsf{ED}}(x, y)$ for any constant $\varepsilon > 0$ with $\tilde{O}_\varepsilon(\sqrt{n})$ space.

Theorem 3.4 ([39]). *Given an offline string y, an online string x, both with length n, and any constant $\varepsilon \in (0, 1]$, there is a deterministic algorithm that outputs a $1 + \varepsilon$ approximation of $\widetilde{\mathsf{ED}}(y, x)$ with $\tilde{O}(\sqrt{n/\varepsilon})$ space in polynomial time.*

Although deletion distance is not equivalent to edit distance we study in this paper, there is a simple transformation that reduces edit distance to deletion distance. The transformation can be found in previous works (see example 6.9 of [120] for example). More specifically, for any given string x, we can obtain a new string x' by prepending a special symbol \$ (\$ is not the in the alphabet set) to each character of x. Thus, say $x = x_1x_2\cdots x_n$, we let $x' = \$x_1\$x_2\cdots\$x_n \in (\Sigma \cup \{\$\})^{2n}$. We can obtain a string y' from y with the same transformation. The following lemma shows that the above transformation reduces computing edit distance to computing deletion distance.

Lemma 3.2.1. *$\widetilde{ED}(x', y') = 2\,ED(x, y)$.*

Proof. We first show that $2\,\mathsf{ED}(x, y) \geq \widetilde{\mathsf{ED}}(x', y')$. Assume we can transform x to y with d edit operations (insertion, deletion, and substitution), we show that it is possible to transform x' to y' with $2d$ insdel operations (insertion and deletion). To see this, if we delete one symbol from x, we delete the corresponding symbol in x' together with the \$ prepended to it. If we insert one symbol to x, we insert the corresponding symbol to x' together with a \$ prepended to it. If we substitute one symbol with another in x, we can first delete the corresponding symbol in x' and insert the new symbol at the same position.

We now show that $2\,\mathsf{ED}(x, y) \leq \widetilde{\mathsf{ED}}(x', y')$. We consider an LCS between x' and y', and consider each pair of adjacent matches of \$ in this LCS. Without loss of generality, we can assume that in at least one side (say x'), this pair looks like $\$a\$$ where a is a symbol (we can also append a \$ at the end if needed), because otherwise if both sides have at least two symbols between the \$'s, then there is another pair of \$'s in the middle that can be matched and added to the LCS. Suppose now in y' there are t non-\$ symbols between the two \$'s. We have two cases: 1. If in the LCS, a is matched to one of the t symbols in y. Then the number of insdel operations between x' and y' we need for this part is $2t - 2$, while for the corresponding part of x and y, we only need $t - 1$ deletions. 2. If in the LCS a is not matched to any of the t symbols in y.

Then the number of insdel operations between x' and y' we need for this part is $2t$, while for x and y we need $t-1$ deletions and one substitution (we can substitute a for any of the t symbols in y), so that's t operations. Thus, if we can transform x' to y' with d insdel operations, we can transform x to y with $\frac{d}{2}$ edit operations.

□

Note that the reduction can be implemented in a streaming manner, thus the following theorem is a direct result of the reduction and Theorem 3.4.

Theorem 3.5. *Given an offline string y, an online string x, both with length n, and any constant $\varepsilon > 0$, there is a deterministic algorithm that outputs a $1+\varepsilon$ approximation of $\mathsf{ED}(y,x)$ with $\tilde{O}(\sqrt{\frac{n}{\varepsilon}})$ space in polynomial time.*

3.2.2 $2^{O(\frac{1}{\delta})}$-Approximation using $O\big(\frac{d^\delta}{\delta}\ \mathrm{polylog}(n)\big)$ Space

We can compute edit distance exactly using dynamic programming with the following recurrence equation. We initialize $A(0,0)=0$ and $A(i,0)=A(0,i)=i$ for $i \in [n]$. Then for $0 \le i,j \le n$,

$$A(i,j) = \begin{cases} A(i-1,j-1), & \text{if } x_i = y_j. \\ \min \begin{cases} A(i-1,j-1)+1, \\ A(i,j-1)+1, \\ A(i-1,j)+1, \end{cases}, & \text{if } x_i \neq y_j. \end{cases} \tag{3.1}$$

Where $A(i,j)$ is the edit distance between $x[1:i]$ and $y[1:j]$. The three options in the case $x_i \neq y_j$ each corresponds to one of the edit operations (substitution, insertion, and deletion). Thus, we can run a dynamic programming to compute matrix A. When the edit distance is bounded by k, we only need to compute the diagonal stripe of matrix A with width $O(k)$. Thus, we have the following.

Lemma 3.2.2. *Assume we are given streaming access to the online string $x \in \Sigma^*$ and random access to the offline string $y \in \Sigma^n$. We can check whether $\mathsf{ED}(x,y) \le k$*

or not, with $O(k \log n)$ bits of space in $O(nk)$ time. If $\mathsf{ED}(x,y) \le k$, we can compute $\mathsf{ED}(x,y)$ exactly with $O(k \log n)$ bits of space in $O(nk)$ time.

Claim 3.2.1. *For two strings $x, y \in \Sigma^n$, assume $\mathsf{ED}(x,y) < k$, we can output the edit operations that transforms x to y with $O(k \log n)$ bits of space in $O(nk^2)$ time.*

Proof. The idea is to run the dynamic programming $O(k)$ times. We initialize $n_1 = n_2 = n$. If $x_{n_1} = y_{n_2}$, we find the largest integer such that $x[n_1 - i : n_1] = y[n_2 - i : n_2]$ and set $n_1 \leftarrow n_1 - i$ and $n_2 \leftarrow n_2 - i$ and continue. If $x_{n_1} \neq y_{n_2}$, we compute the the elements $A(n_1, n_2 - 1)$, $A(n_1 - 1, n_2 - 1)$, and $A(n_1 - 1, n_2)$. By 3.1, we know there is some $(n_1', n_2') \in \{(n_1, n_2 - 1), (n_1 - 1, n_2 - 1), (n_1 - 1, n_2)\}$ such that $A(n_1', n_2') = A(n_1, n_2) - 1$. We set $n_1 \leftarrow n_1'$ and $n_2 \leftarrow n_2'$, output the corresponding edit operation and continue. We need to run the dynamic programming $O(k)$ times. The running time is $O(nk^2)$. □

Lemma 3.2.3. *For two strings $x, y \in \Sigma^n$, assume $d = \mathsf{ED}(x,y)$. For any integer $1 \le t \le d$, there is a way to divide x and y each into t parts so that $x = x^1 \circ x^2 \circ \cdots \circ x^t$ and $y = y^1 \circ y^2 \circ \cdots \circ y^t$ (we allow x^ior y^i to be empty for some i), such that $d = \sum_{i=1}^{t} \mathsf{ED}(x^i, y^i)$ and $\mathsf{ED}(x^i, y^i) \le \lceil \frac{d}{t} \rceil$ for all $i \in [t]$.*

Proof of Lemma 3.2.3. Since $\mathsf{ED}(x,y) = d$, we can find d edit operations on x that transforms x into y. We can write these edit operations in the same order as where they occured in x. Then, we first find the largest i_1 and j_1, such that the first $\lceil \frac{d}{t} \rceil$ edit operations transforms $x[1 : i_1]$, to $y[1 : j_1]$. Notice that i_1 (or j_1) is 0 if the first $\lceil \frac{d}{t} \rceil$ edit operations insert $\lceil \frac{d}{t} \rceil$ before x^1 (or delete first $\lceil \frac{d}{t} \rceil$ symbols in x). We can set $x^1 = x[1 : i_1]$ and $y^1 = y[1 : j_1]$ and continue doing this until we have seen all d edit operations. This will divide x and y each in to at most t parts. □

We now present the algorithm. The main ingredient of our algorithm is a recursive procedure called $\mathsf{FindLongestSubstring}$. The pseudocode is given in Algorithm 10. It takes four inputs: an online string x, an offline string y, an upper bound of edit distance u, and an upper bound of space available s. The output of FindLongestSubstring is a three tuple: a pair of indices (p, q), two integers l and d. Througout the analysis, we assume ε is a small constant up to our choice.

Algorithm 10: FindLongestSubstring

```
Data: An online $x \in \Sigma^*$ with streaming access, an local string $y \in \Sigma^n$, an
      upper bound of edit distance $u$, and an upper bound of space available
      $s$.
 1: if $u \le s$ then
 2:    for $l' = u$ to $|x|$ do
 3:       try all $1 \le p' < q' \le n$ such that $l' - u \le |q' - p' + 1| \le l' + u$ and check
          whether $\mathsf{ED}(y[p' : q'], x[1 : l']) \le u$. ;
 4:       if there exists a pair $p', q'$ such that $\mathsf{ED}(y[p' : q'], x[1 : l']) \le u$ then
 5:          set $p, q$ to be the pair of indices that minimizes $\mathsf{ED}(y[p' : q'], x[1 : l'])$.;
 6:          $d \leftarrow \mathsf{ED}(y[p : q], x[1 : l'])$, and $l \leftarrow l'$. ;
 7:          compute and record the edit operations that transform $y[p : q]$ to $x[1 : l']$. ;
 8:          continue
 9:       else
10:          break
11: else
12:    initialize $i = 1$, $a_i = 1$, $l = 0$. ;
13:    while $a_i \le |x|$ and $i \le s$ do
14:       $(p_i, q_i), l_i, d_i \leftarrow$ FindLongestSubstring$(x[a_i : n], y, \lceil u/s \rceil, s)$. ;
15:       $i \leftarrow i + 1$. ;
16:       $a_i \leftarrow a_{i-1} + l_i$. ;
17:       $l \leftarrow l + l_i$.;
18:    $T \leftarrow i - 1$. ;
19:    for all $1 \le p' < q' \le n$, use the algorithm guaranteed by Theorem 2.1 to
       compute $\tilde{d}(p', q')$, a $1 + \varepsilon$ approximation of
       $\mathsf{ED}(y[p' : q'], y[p_1 : q_1] \circ y[p_2 : q_2] \circ \cdots \circ y[p_T : q_T])$. ;
20:    $p, q \leftarrow \mathbf{argmin}_{p',q'} \tilde{d}(p', q')$.              ▷ $p, q$ minimizes $\tilde{d}$;
21:    $d \leftarrow \tilde{d}(p, q) + \sum_{i=1}^{T} d_i$. ;
22: return $(p, q)$, $l$, $d$.
```

We have the following Lemma.

Lemma 3.2.4. *Let $(p, q), l, d$ be the output of* FindLongestSubstring *with input x, y, u, s. Then assume l^0 is the largest integer such that there is a substring of y, say $y[p^0 : q^0]$, with $\mathsf{ED}(x[1 : l^0], y[p^0 : q^0]) \le u$. Then, we have $l \ge l^0$. Also, assume $y[p^* : q^*]$ is the* *substring of y that is closest to $x[1:l]$ in edit distance. We have*

$$\mathit{ED}\left(x[1:l], y[p^*:q^*]\right) \leq \mathit{ED}\left(x[1:l], y[p:q]\right) \leq d \leq c(u,s)\, \mathit{ED}\left(x[1:l], y[p^*:q^*]\right) \tag{3.2}$$

where $c(u,s) = 2^{O(\log_s u)}$ if $u > s$ and $c(u,s) = 1$ if $u \leq s$

Proof of Lemma 3.2.4. In the following, $(p,q), l, d$ are the output of FindLongestSubstring with input x, y, u, s. We let l^0 be the largest integer such that there is a substring of y, say $y[p^0:q^0]$, with $\mathsf{ED}(x[1:l^0], y[p^0:q^0]) \leq u$ and $y[p^*:q^*]$ is the substring of y that minimizes the edit distance to $x[1:l]$.

When $u \leq s$, we first set $l' = u$, then we know for any $p, q \in [n]$ such that $q - p + 1 = u$, $\mathsf{ED}\left(x[1:l'], y[p:q]\right) \leq u$ since we can transform $x[1:l']$ to $y[p:q]$ with u substitutions. Thus, we are guaranteed to find such a pair (p,q) and we can record the edit operation that transform $y[p:q]$ to $x[1:l]$. We set $l = l'$, $d = \mathsf{ED}(x[1:l'], y[p:q])$ and continue with $l' \leftarrow l' + 1$. When $l' > u$ $(l = l' - 1)$. we may not be able to store $x[1:l]$ in the memory for random access since our algorithm uses at most $O(s \log n)$ bits of space. However, we have remembered a pair of indices (p,q) and at most u edit operations that can transform $y[p:q]$ to $x[1:l]$. This allows us to query each bit of $x[1:l]$ from left to right once with $O(u+l)$ time. Thus, for each substring $y[p:q]$ of y, we can compute its edit distance from $x[1:l']$. Once we find such a substring with $\mathsf{ED}(x[1:l'], y[p:q]) \leq u$, by Claim 3.2.1, we can then compute the edit operations that transfom $y[p,q]$ to $x[1:l']$ with $O(u \log n)$ space. Thus, if $u \leq s$, we can find the largest integer l such that there is a substring of y, denoted by $y[p:q]$, with $\mathsf{ED}\left(x[1:l], y[p:q]\right) = u$ with $O(s \log n)$ bits of space. If there is no substring $y[p:q]$ with $\mathsf{ED}\left(x[1:l'], y[p:q]\right) \leq u$, we terminate the procedure and return current (p,q), l, and d.

If $l^0 > l$, then $x[1:l+1]$ is a substring of $x[1:l^0]$, we can find a substring of $y[p^0:q^0]$ such that its edit distance to $x[1:l+1]$ is at most u. Thus, FindLongestSubstring will not terminate at l. We must have $l = l^0$.

Also notice that when $u \le s$, we always do exact computation. $y[p : q]$ is the substring in y that minimizes the edit distance to $x[1 : l]$. Thus, Lemma 3.2.4 is correct when $u \le s$.

For the case $u > s$, FindLongestSubstring needs to call itself recursively. Notice that each time the algorithm calls itself and enters the next level, the upper bound of edit distance u is reduced by a factor s. The recursion ends when $u \le s$. Thus, we denote the depth of recursion by d, where $d = O(\log_s u)$. We assume the algoirthm starts from level 1 and at level i for $i \le d$, the upper bound of edit distance becomes $\frac{u}{s^{i-1}}$.

We prove the Lemma by induction on the depth of recursion. The base case of the induction is when $u \le s$, for which we have shown the correctness of Lemma 3.2.4. We now assume Lemma 3.2.4 holds when the input is $x, y, \lceil u/s \rceil, s$ for any strings x, y.

We first show $l \ge l^0$. Notice that the while loop at line 13 terminates when either $a_i > |x|$ or $i > s$. If $a_i > |x|$, we know l is set to be $a_i - 1 = |x|$. Since $l^0 \le |x|$ by definition. We must have $l \ge l^0$.

If the while loop terminates when $i > s$. By the definition of l^0, we know $x[1 : l^0]$ and $y[p^0 : q^0]$ can be divided into s blocks, let

$$\begin{aligned} x[1 : l^0] &= x[1 : l_1^0] \circ x[l_1^0 + 1 : l_1^0 + l_2^0] \circ \cdots \circ x[\sum_{i=1}^{s-1} l_i^0 + 1 : \sum_{i=1}^{s} l_i^0] \\ y[p^0 : q^0] &= y[p_1^0 : q_1^0] \circ y[p_2^0 : q_2^0] \circ \cdots \circ y[p_s^0 : q_s^0] \end{aligned}$$

where $l^0 = \sum_{i=1}^{s} l_i^0$. We have

$$\mathsf{ED}\left(x[\sum_{j=1}^{i-1} l_j^0 + 1 : \sum_{j=1}^{i} l_j^0], y[p_i^0 : q_i^0]\right) \le \lceil u/s \rceil, \quad \forall i \in [s].$$

For convenience, we denote $b_i^0 = \sum_{j=1}^{i} l_j^0$ and $b_i = \sum_{j=1}^{i} l_j$. By the defintion, we know $b_s^0 = l^0$ and $b_s = l$ and all l_i^0, l_i are non-negative. We have $b_i = a_{i+1} - 1$.

We show that $b_i \ge b_i^0$ for all $i \in [s]$ by induction on i. For the base case $i = 1$, let $\overline{l}_1^0$ be the largest integer such that there is a substring of y within edit distance $\lceil u/s \rceil$ to $x[1 : \overline{l}_1^0]$. We know $\overline{l}_1^0 \geq l_1^0$. Since we assume Lemma 3.2.4 holds for inputs $x, y, \lceil u/s \rceil, s$, we know $l_1 \geq \overline{l}_1^0$. Thus, $l_1 \geq l_1^0$ and $b_1 \geq b_1^0$. Now assume $b_i \geq b_i^0$ holds for some $i \in [s-1]$, we show that $b_{i+1} \geq b_{i+1}^0$. If $b_i \geq b_{i+1}^0$, $b_{i+1} \geq b_{i+1}^0$ holds. We can assume $b_{i+1}^0 > b_i \geq b_i^0$. We show that $l_{i+1} \geq b_{i+1}^0 - b_i$. To see this, let $\overline{l}_i^0$ be the largest integer such that there is a substring of y within edit distance $\lceil u/s \rceil$ to $x[b_i + 1 : b_i + \overline{l}_{i+1}^0]$. We know $l_{i+1} \geq \overline{l}_{i+1}^0$ since we assume Lemma 3.2.4 holds for inputs $x[b_i + 1 : |x|], y, \lceil u/s \rceil, s$. Notice that $\mathsf{ED}(x[b_i^0 + 1 : b_{i+1}^0], y[p_i^0, q_i^0]) \leq \lceil u/s \rceil$, we know $\overline{l}_i^0$ is at least $b_{i+1}^0 - b_i$ since $x[b_i + 1 : b_{i+1}^0]$ is a substring of $x[b_i^0 + 1 : b_{i+1}^0]$. We know $l_{i+1} \geq \overline{l}_{i+1}^0 \geq b_{i+1}^0 - b_i$. Thus, $b_{i+1} = b_i + l_{i+1} \geq b_{i+1}^0$. This proves $l \geq l_0$.

We now prove inequality 3.2. After the while loop, the algorithm then finds a substring of y, $y[p : q]$, that minimizes $\tilde{d}(p', q')$ where $\tilde{d}(p', q')$ is a $1 + \varepsilon$ approximation of $\mathsf{ED}(y[p' : q'], y[p_1 : q_1] \circ y[p_2 : q_2] \circ \cdots \circ y[p_T : q_T])$. For convenience, we denote

$$\tilde{y} = y[p_1 : q_1] \circ y[p_2 : q_2] \circ \cdots \circ y[p_T : q_T].$$

Thus,

$$\mathsf{ED}\left(y[p : q], \tilde{y}\right) \leq \tilde{d}(p, q) \leq (1 + \varepsilon) \cdot \mathsf{ED}\left(y[p^* : q^*], \tilde{y}\right). \tag{3.3}$$

Let $y[\bar{p}_j^* : \bar{q}_j^*]$ be the substring of y that is closest to $x[a_j : a_{j+1} - 1]$ in edit distance. By the inductive hypothesis, we assume the output of $\mathsf{FindLongestSubstring}(x, y, \lceil u/s \rceil, s)$ satisfies Lemma 3.2.4. We know

$$\begin{aligned} \mathsf{ED}\left(x[a_j : a_{j+1} - 1], y[\bar{p}_j^* : \bar{q}_j^*]\right) &\leq \mathsf{ED}\left(x[a_j : a_{j+1} - 1], y[p_j : q_j]\right) \\ &\leq d_j \\ &\leq c(u/s, s)\, \mathsf{ED}\left(x[a_j : a_{j+1} - 1], y[\bar{p}_j^* : \bar{q}_j^*]\right). \end{aligned} \tag{3.4}$$

By the optimality of $y[p^* : q^*]$ and triangle inequality, we have

$$\begin{aligned}
\mathsf{ED}\left(x[1:l], y[p^*:q^*]\right) &\le \mathsf{ED}\left(x[1:l], y[p:q]\right) \\
&\le \mathsf{ED}\left(x[1:l], \tilde{y}\right) + \mathsf{ED}\left(\tilde{y}, y[p:q]\right) \quad \text{(By triangle inequality)} \\
&\le \sum_{i=1}^{T} d_i + \tilde{d}(p,q) \\
&= d.
\end{aligned}$$

Also notice that we can write $y[p^*:q^*] = y[p_1^*:q_1^*] \circ y[p_2^*:q_2^*] \circ \cdots \circ y[p_T^*:q_T^*]$ such that

$$\mathsf{ED}\left(x[1:l], y[p^*:q^*]\right) = \sum_{i=1}^{T} \mathsf{ED}\left(x[a_i:a_{i+1}+1], y[p_i^*, q_i^*]\right). \tag{3.5}$$

We have

$$\begin{aligned}
d &= \sum_{i=1}^{T} d_j + \tilde{d}(p,q) \\
&\le \sum_{i=1}^{T} d_i + (1+\varepsilon)\,\mathsf{ED}\left(\tilde{y}, y[p^*:q^*]\right) && \text{By 3.3} \\
&\le \sum_{i=1}^{T} d_i + (1+\varepsilon)\Big(\mathsf{ED}\left(x[1:l], y[p^*:q^*]\right) + \mathsf{ED}\left(x[1:l], \tilde{y}\right)\Big) && \text{(By triangle inequality)} \\
&\le \sum_{i=1}^{T} d_i + (1+\varepsilon)\sum_{i=1}^{T} \mathsf{ED}\left(x[a_i:a_{i+1}-1], y[p_i^*:q_i^*]\right) \\
&\quad + (1+\varepsilon)\sum_{i=1}^{T} \mathsf{ED}\left(x[a_i:a_{i+1}-1], y[p_i:q_i]\right) \\
&\le (1+\varepsilon)\cdot \sum_{i=1}^{T}\Big(\big(2c(u/s,s)+1\big)\,\mathsf{ED}\left(x[a_i:a_{i+1}-1], y[p_i^*:q_i^*]\right)\Big) && \text{By 3.4} \\
&\le (1+\varepsilon)\big(2c(u/s,s)+1\big)\cdot \mathsf{ED}\left(x[1:l], y[p^*:q^*]\right) && \text{By 3.5}
\end{aligned}$$

We set $c(u,s) = (1+\varepsilon)\big(2c(u/s,s)+1\big)$. Since we assume $c(u/s,s) = 2^{O(\log_s(u/s))}$, we know $c(u,s) = 2^{O(\log_s u)}$. This proves inequality 3.2.

□

Lemma 3.2.5. *Given any $x, y \in \Sigma^n$, let $u, s \leq n$,* $\mathsf{FindLongestSubstring}(x, y, u, s)$ *runs in polynomial time. It queries x from left to right in one pass and uses $O\Big(s \log_s u \cdot \text{polylog}(n)\Big)$ space.*

Proof of Lemma 3.2.5. If $u \leq s$, we need to store at most u edit operations that transforms, $y[p : q]$ to $x[1 : l]$ for current (p, q) and l. This takes $O(u \log n) = O(s \log n)$ bits of space. Notice when $l \geq u$, we do not need to remember $x[1 : l]$. Instead, we query $x[1 : l]$ by looking at $y[p : q]$ and the edit operations.

If $u \geq s$, the algorithm is recursive. Let us consider the recursion tree. We assume the algoirthm starts from level 1 (root of the tree) and the depth of the recursion tree is d. At level i for $i \leq d$, the upper bound of edit distance (third input to algorithm $\mathsf{FindLongestSubstring}$) is $u_i = \frac{u}{s^{i-1}}$. The recursion stops when $u_i \leq s$. Thus, the depth of recursion d is $O(\log_s u)$ by our assumption on u and s. The order of computation on the recursion tree is the same as depth-first search and we only need to query x at the bottom level. There are at most $s^d = O(u)$ leaf nodes (nodes at the bottom level). For the i-th leaf nodes of the recursion tree, we are computing $\mathsf{FindLongestSubstring}(x[a_i : n], y, u_d, s)$ with $u_d \leq s$ where a_i is the last position visited by the previous leaf node. Thus, we only need to access x from left to right in one pass.

For each inner node, we need to remember s pairs of indices (p_i, q_i) and s integers d_i for $i \in [s]$, which takes $O(s \log n)$ space. For computing an $(1 + \varepsilon)$-approximation of $\tilde{d}(p', q')$, we can use the space-efficient approximation algorithm from Chapter 2 which uses only polylog(n) space. Thus, each inner node takes $O(s \ \text{polylog}(n))$ bits of space. For the leaf nodes, we have $u \leq s$. Thus, we can compute it with $O(s \log n)$ bits of extra memory. Since the order of computation is the same as depth-first search, we only need to maintain one node in each recursive level and we can reuse the space for those nodes we have already explored. Since the depth of recursion is $O(\log_s u)$, the total space required is $O\Big(s\log_s u \cdot \mathrm{polylog}(n)\Big)$.

For the time complexity, notice that the space efficient algorithm for $1+\varepsilon$ approximating ED takes polynomial time. The depth of recursion is $O(\log_s u)$ and at each recursive level, the number of nodes is polynomial in n , we need to try $O(n^2)$ different p', q' at each node except the leaf nodes. Thus, the running time is still polynomial.

□

We now present our algorithm for approximating edit distance in the asymmetric streaming model. The pseudocode of our algorithm is given in algorithm 11. It takes three input, an online string x, an offline string y and a parametert $\delta \in (0, 1/2]$.

Algorithm 11: AsymED: asymmetric streaming algorithm for ED

```
Data: Two strings: $x \in \Sigma^*$ and $y \in \Sigma^n$, a constant $\delta \in (0, 1/2]$
 1: initialize $a \leftarrow 1$, $i \leftarrow 1$.
 2: set $k$ to be a large constant such that $s = k^\delta$ is an integer.
 3: while $a \le n$ do
 4:     $(p_i, q_i), l_i, d_i \leftarrow$ FindLongestSubstring$(x[a:n], y, k/s, s)$.
 5:     $a \leftarrow a + l_i$.
 6:     $i \leftarrow i + 1$.
 7:     if $i \ge k^\delta$ then
 8:         $k \leftarrow 2^{1/\delta} k$.
 9:         $s \leftarrow k^\delta$.
10: $T \leftarrow i - 1$.
11: compute $\tilde{d}$, an $(1+\varepsilon)$-approximation of ED$(y, y[p_1, q_1] \circ y[p_2, q_2] \circ \cdots \circ y[p_T, q_T])$.
    ▷ Using the algorithm guaranteed by Theorem 2.1
12: return $\bar{d} = \tilde{d} + \sum_{i=1}^{T} d_i$.
```

Lemma 3.2.6. *Assume $d =$ ED(x, y), Algorithm 11 can be run with $O(\frac{d^\delta}{\delta}\,\mathrm{polylog}(n))$ bits of space in polynomial time.*

Proof. Notice that k is initially set to be a constant k_0 such that $s_0 = k_0^\delta$ is an integer. k is multiplied by $2^{1/\delta}$ whenever $i \ge k$. We assume that in total, k is updated u times and after the j-th update, $k = k_j$, where $k_j = 2^{1/\delta} k_{j-1}$. We let $s_j = k_j^\delta$. Thus,

$k^\delta_{j+1} = 2k^\delta_j$ and $s_{j+1} = 2s_j$. We denote the a before i-th while loop by a_i so that $a_1 = 1$ and $a_i = 1 + \sum_{j=1}^{i-1} l_j$ for $1 < i \le T$.

We first show the following claim.

Claim 3.2.2. $k^\delta_u \le 8d^\delta$.

Proof. Assume the contrary, we have $k^\delta_u > 8d^\delta$, and thus $k^\delta_{u-1} > 4d^\delta$.

Let $i_0 = k^\delta_{u-2}$. That is, after i is updated to i_0, k is updated from k_{u-2} to k_{u-1}. For convenience, we denote $\bar{x} = x[a_{i_0} : n]$. Since $\mathsf{ED}(x, y) \le d$, there is a substring of y, say $\bar{y}$, such that $\mathsf{ED}(\bar{x}, \bar{y}) \le d$. Let $\xi = \frac{k_{u-1}}{s_{u-1}}$, by Lemma 3.2.3, we can partition $\bar{x}$ and $\bar{y}$ each into $\lceil d/\xi \rceil$ parts such that

$$\bar{x} = \bar{x}^1 \circ \bar{x}^2 \circ \cdots \circ \bar{x}^{\lceil d/\xi \rceil},$$
$$\bar{y} = \bar{y}^1 \circ \bar{y}^2 \circ \cdots \circ \bar{y}^{\lceil d/\xi \rceil},$$

and $\mathsf{ED}(\bar{x}^j, \bar{y}^j) \le \xi$ for $j \in [\lceil d/\xi \rceil]$. We denote $\bar{x}^j = x[\beta_j : \gamma_j]$ for $j \in [\lceil d/\xi \rceil]$ so that $\bar{x}^j$ starts with β_j-th symbol and ends with γ_j-th symbol of x. We have $\beta_1 = a_{i_0}$ and $\gamma_i + 1 = \beta_{i+1}$.

We first show that for $i_0 \le i \le T$, $a_i \ge \beta_{i-i_0+1}$. Assume there is some i such that $a_i \ge \beta_{i-i_0+1}$ and $a_{i+1} < \beta_{i-i_0+2}$. By Lemma 3.2.4, l_i is at least the largest integer l such that there is a substring of y within edit distance ξ to $x[a_i : a_i + l - 1]$. Since $x[a_i : a_{i+1} - 1] = x[a_i : a_i + l_i - 1]$ is a substring of $x[\beta_{i-i_0+1} : \gamma_{i-i_0+1}] = \bar{x}^{i-i_0+1}$ and $\mathsf{ED}(\bar{x}^{i-i_0+1}, \bar{y}^{i-i_0+1}) \le \xi$. There is a substring of y within edit distance ξ to $x[a_i : \gamma_{i-i_0+1}]$. Thus, we must have $l_i \ge \gamma_{i-i_0+1} - a_i + 1$. Thus $a_{i+1} = a_i + l_i \ge \gamma_{i-i_0+1} + 1 = \beta_{i-i_0+2}$. This is contradictory to our assumption that $a_{i+1} < \beta_{i-i_0+2}$.

We now show that $T \le i_0 + \lceil d/\xi \rceil - 1$. If $T > i_0 + \lceil d/\xi \rceil$, we have $a_{i_0+\lceil d/\xi \rceil - 1} \ge \beta_{\lceil d/\xi \rceil}$. By Lemma 3.2.4, we must have $a_{i_0+\lceil d/\xi \rceil} = n + 1$. Since $a_{i_0+\lceil d/\xi \rceil} > n$, we will terminate the while loop and set $T = i_0 + \lceil d/\xi \rceil - 1$. Thus, we have $T \le i_0 + \lceil d/\xi \rceil - 1$.

Meanwhile, by the assumption that $k^\delta_{u-1} > 4d^\delta$, we have $k^\delta_{u-1} - k^\delta_{u-2} = k^\delta_{u-2} > 2d^\delta >$

$2d/\xi$. If k is updated u times, T is at least $k_{u-1}^{\delta}-1$. Thus, $T \geq k_{u-1}^{\delta}-1 > i_0+2d/\xi-1$. This is contradictory to $T \leq i_0 + \lceil d/\xi \rceil - 1$. □

By the above claim, we have $T = O(d^{\delta})$. Thus, we can remember all $(p_i, q_i), l_i, d_i$ for $i \in [T]$ with $O(d^{\delta} \log n)$ bits of space. For convenience, let $\tilde{y} = y[p_1 : q_1] \circ y[p_2 : q_2] \circ \cdots \circ y[p_T : q_T]$. We can use the space efficient algorithm from Chapter 2 (Theorem 2.1) to compute a $(1+\varepsilon)$-approximation of $\mathsf{ED}(y, \tilde{y})$ with $\text{polylog}(n)$ bits of space in polynomial time. By Lemma 3.2.5, we can run FindLongestSubstring with $O(\frac{d^{\delta}}{\delta}\,\text{polylog}(n))$ space since $s = O(d^{\delta})$. The total amount of space used is $O(\frac{d^{\delta}}{\delta}\,\text{polylog}(n))$.

We run FindLongestSubstring $O(d^{\delta})$ times and compute a $(1+\varepsilon)$-approximation of $\mathsf{ED}(y, \tilde{y})$ with polynomial time. The total running time is still a polynomial. □

Lemma 3.2.7. *Assume $\mathsf{ED}(x, y) = d$, there is a one pass deterministic algorithm that outputs a $(3+\varepsilon)$-approximation of $\mathsf{ED}(x, y)$ in asymmetric streaming model, with $O(\sqrt{d}\ \text{polylog}(n))$ bits of space in polynomial time.*

Proof of Lemma 3.2.7. We run algorithm 11 with parameter $\delta = 1/2$. The time and space complexity follows from Lemma 3.2.6.

Notice that algorithm 11 executes FindLongestSubstring T times and records T outputs $(p_i, q_i), l_i, d_i$ for $i \in [T]$. We also denote $a_i = 1 + \sum_{j=1}^{i-1} l_j$ with $a_1 = 1$. Thus, we can partition x into T parts such that

$$x = x^1 \circ x^2 \circ \cdots \circ x^T$$

where $x^i = x[a_i : a_{i+1} - 1]$. Since $s = k^{1/2} = k/s$, by Lemma 3.2.4, we know $\mathsf{ED}(x^i, y[p_i : q_i]) = d_i$.

We now show that the output $\bar{d} = \tilde{d} + \sum_{i=1}^{T} d_i$ is a $3+\varepsilon$ approximation of $d = \mathsf{ED}(x, y)$. Let $\tilde{y} = y[p_1 : q_1] \circ y[p_2 : q_2] \circ \cdots \circ y[p_T : q_T]$. Notice that $\mathsf{ED}(x, \tilde{y}) \leq \sum_{i=1}^{T} \mathsf{ED}(x^i, y[p_i : q_i])$ and $\mathsf{ED}(\tilde{y}, y) \leq \tilde{d}$. We have

$$\begin{aligned}\mathsf{ED}(x,y) &\le \mathsf{ED}(x,\tilde{y}) + \mathsf{ED}(\tilde{y},y) && \text{By triangle inequality}\\ &\le \tilde{d} + \sum_{i=1}^{T} d_i\\ &= \bar{d}\end{aligned}$$

On the other hand, we can divide y into T parts such that $y = \hat{y}^1 \circ \hat{y}^2 \circ \cdots \circ \hat{y}^T$ and guarantee that

$$\mathsf{ED}(x,y) = \sum_{i=1}^{T} \mathsf{ED}(x^i, \hat{y}^i). \tag{3.6}$$

Also, by Lemma 3.2.4, $y[p_i : q_i]$ is the substring of y that is closest to x^i in edit distance, we know

$$\mathsf{ED}(x^i, \hat{y}^i) \ge \mathsf{ED}(x^i, y[p_i : q_i]). \tag{3.7}$$

We have

$$\begin{aligned}\bar{d} &= \tilde{d} + \sum_{i=1}^{T} d_i\\ &\le (1+\varepsilon)\,\mathsf{ED}(y,\tilde{y}) + \sum_{i=1}^{T} \mathsf{ED}\left(x^i, y[p_i : q_i]\right)\\ &\le (1+\varepsilon)\Big(\mathsf{ED}(y,x) + \mathsf{ED}(\tilde{y},x)\Big) + \sum_{i=1}^{T} \mathsf{ED}\left(x^i, y[p_i : q_i]\right)\\ &\le (1+\varepsilon)\sum_{i=1}^{T} \mathsf{ED}\left(x^i, \hat{y}^i\right) + (2+\varepsilon)\sum_{i=1}^{T} \mathsf{ED}\left(x^i, y[p_i : q_i]\right) && \text{By 3.6}\\ &\le (3+2\varepsilon)\sum_{i=1}^{T} \mathsf{ED}\left(x^i, \hat{y}^i\right) && \text{By 3.7}\\ &= (3+2\varepsilon)\,\mathsf{ED}(x,y).\end{aligned}$$

Since we can pick ε to be any constant, we pick $\varepsilon' = 2\varepsilon$ and the output d is indeed a $3+\varepsilon'$ approximation of $\mathsf{ED}(x,y)$. □

Lemma 3.2.8. *Assume $ED(x,y) = d$, for any constant $\delta \in (0, 1/2)$, there is an algorithm that outputs a $2^{O(\frac{1}{\delta})}$-approximation of $ED(x,y)$ with $O(\frac{d^\delta}{\delta}\,\mathrm{polylog}(n))$ bits of space in polynomial time in the asymmetric streaming model.*

Proof of Lemma 3.2.8. We run algorithm 11 with parameter $\delta \in (0, 1/2)$. Without loss of generality,we can assume $1/\delta$ is an integer. The time and space complexity follows from Lemma 3.2.6.

Again we can divide x into T parts such that $x^i = x[a_i : a_{i+1} - 1]$ where $a_i = 1 + \sum_{j=1}^{i-1} l_j$. By Lemma 3.2.4, we have $\mathsf{ED}(x^i, y[p_i : q_i]) \leq d_i$.

Let $y = \hat{y}^1 \circ \hat{y}^2 \circ \cdots \circ \hat{y}^T$ be a partition of y such that

$$\mathsf{ED}(x, y) = \sum_{i=1}^{T} \mathsf{ED}\left(x^i, \hat{y}^i\right).$$

We now show that $\bar{d}$ is a $2^{O(1/\delta)}$ approximation of $d = \mathsf{ED}(x, y)$. Similarly, we let $\tilde{y} = y[p_1 : q_1] \circ y[p_2 : q_2] \circ \cdots \circ y[p_T : q_T]$. We have

$$\begin{aligned} \mathsf{ED}(x, y) &\leq \mathsf{ED}(x, \tilde{y}) + \mathsf{ED}(\tilde{y}, y) && \text{By triangle inequality} \\ &\leq \tilde{d} + \sum_{i=1}^{T} d_i \\ &= \bar{d} \end{aligned}$$

Let $y[p_i^* : q_i^*]$ be the substring of y that is closest to x^i in edit distance. By Lemma 3.2.4, we know

$$\mathsf{ED}(x^i, y[p_i^* : q_i^*]) \leq \mathsf{ED}(x^i, y[p_i : q_i]) \leq c\,\mathsf{ED}(x^i, y[p_i^* : q_i^*]) \tag{3.8}$$

where $c = 2^{O(\log_s(k^{1-\delta}))} = 2^{O(1/\delta)}$. Thus, we have

$$\mathsf{ED}\left(x^i, y[p_i : q_i]\right) \leq c\,\mathsf{ED}(x^i, y[p_i^* : q_i^*]) \leq c\,\mathsf{ED}\left(x^i, \hat{y}^i\right). \tag{3.9}$$

Thus, we can get a similar upper bound of $\bar{d}$ by

$$\begin{aligned}
\bar{d} &= \tilde{d} + \sum_{i=1}^{T} d_i \\
&\leq (1+\varepsilon)\,\mathsf{ED}(y,\tilde{y}) + \sum_{i=1}^{T} \mathsf{ED}\left(x^i, y[p_i : q_i]\right) \\
&\leq (1+\varepsilon)\Big(\mathsf{ED}(y,x) + \mathsf{ED}(\tilde{y},x)\Big) + \sum_{i=1}^{T} \mathsf{ED}\left(x^i, y[p_i : q_i]\right) \\
&\leq (1+\varepsilon)(1+2c)\sum_{i=1}^{T} \mathsf{ED}\left(x^i, \hat{y}^i\right) && \text{By 3.9} \\
&= 2^{O(1/\delta)}\,\mathsf{ED}(x,y).
\end{aligned}$$

This finishes the proof. □

Theorem 3.2 is a direct result of Lemma 3.2.7 and Lemma 3.2.8.

3.3 Longest Common Subsequence

3.3.1 $(1+\varepsilon)$-Approximation using $\tilde{O}\big(\frac{\sqrt{n}}{\varepsilon}\big)$ Space.

Below is a formal proof of the longest common subsequence part of Theorem 3.1.

Proof of Theorem 3.1 (The longest common subsequence part). In Algorithm 8, we take $b = \sqrt{n}$ and ε' to be a constant sufficiently smaller than the given positive constant ε. We will show $\mathsf{ApproxLCS}(x, y, \sqrt{n}, \varepsilon')$ works in the asymmetric streaming model.

Our algorithm $\mathsf{ApproxLCS}$ is essentially computing the length of LIS of $z = \mathsf{ReduceLCStoLIS}(x, y)$. In that algorithm, z is naturally divided into b blocks $z = z^1 \circ z^2 \circ \cdots \circ z^b$, where $z^i = \mathsf{ReduceLCStoLIS}(x^i, y)$.

When $b = \sqrt{n}$, the recursion has depth only 2. We need to query x when running $\mathsf{PatienceSorting}$ on z^i at the second level of recursion. Also notice that to read z^i, we only needs to query the i-th block x^i. By Lemma 2.4.2, $\mathsf{ApproxLCS}(x, y, b, \varepsilon')$ can be computed in $\tilde{O}(n^{\frac{5}{2}})$ time with $O(\frac{\sqrt{n}}{\varepsilon}\log n)$ bits of space.

Notice that PatienceSorting only need to read the input string from left to right once. Also, to read $z_i = \mathsf{ReduceLCStoLIS}(x^i, y)$ from left to right once, we only need to scan x^i from left to right once. Thus, ApproxLCS makes only one pass through x when $b = \sqrt{n}$. $\square$

3.3.2 An Algorithm for Binary Strings

We now proof Theorem 3.3. We show that the algorithm presented in [11] for approximating LCS to $(1/2 + \varepsilon)$ factor can be slightly modified to work in the asymmetric streaming model.

The algorithm from [11] uses four algorithms as ingredients Match, BestMatch, Greedy, ApxED (to distinguish from ApproxED). We give a description here and show why they can be modified to work in the asymmetric streaming model.

Algorithm Match takes three inputs: two strings $x, y \in \Sigma^*$ and a symbol $\sigma \in \Sigma$. Let $\sigma(x)$ be the number of symbol σ in string x and we similarly define $\sigma(y)$. $\mathsf{Match}(x, y, \sigma)$ computes the length of longest common subsequence between x and y that consists of only σ symbols. Thus, $\mathsf{Match}(x, y, \sigma) = \min(\sigma(x), \sigma(y))$.

Algorithm BestMatch takes two strings $x, y \in \Sigma^*$ as inputs. It is defined as $\mathsf{BestMatch}(x, y) = \max_{\sigma\in\Sigma} \mathsf{Match}(x, y, \sigma)$.

Algorithm Greedy takes three strings $x^1, x^2, y \in \Sigma^*$ as inputs. It finds the optimal partition $y = y^1 \circ y^2$ that maximizes $\mathsf{BestMatch}(x^1, y^1) + \mathsf{BestMatch}(x^2, y^2)$. The output is $\mathsf{BestMatch}(x^1, y^1) + \mathsf{BestMatch}(x^2, y^2)$ and y^1, y^2.

Algorithm ApxED takes two strings $x, y \in \Sigma^n$ as inputs. Here we require the input strings have equal length. ApxED first computes a constant approximation of $\mathsf{ED}(x, y)$, denoted by $\tilde{\mathsf{ED}}(x, y)$. The output is $n - \tilde{\mathsf{ED}}(x, y)$.

In the following, we assume $\Sigma = \{0, 1\}$ and input strings to the main algorithm x, y both have length n. All functions are normalized with respect to n. Thus, we have $1(x)$,$0(x)$,$\mathsf{LCS}(x,y)$, $\mathsf{ApxED}(x,y) \in (0,1)$.

The algorithm first reduce the input to the perfectly unbalanced case.

We first introduce a few parameters.

$\alpha = \min\{1(x), 1(y), 0(x), 0(y)\}$.

$\beta = \theta(\alpha)$ is a constant. It can be smaller than α by an arbitrary constant factor.

$\gamma \in (0,1)$ is a parameter that depends on the accuracy of the approximation algorithm for $\mathsf{ED}(x,y)$ we assume.

Definition 3.3.1 (Perfectly unbalanced)**.** *We say two string $x, y \in \Sigma^n$ are perfectly unbalanced if*

$$|1(x) - 0(y)| \le \delta\alpha, \tag{3.10}$$

and

$$0(x) \notin [1/2 - \beta', 1/2 + \beta']. \tag{3.11}$$

Here, we require δ to be a sufficiently small constant such that $\delta\alpha \le \beta$ and $\beta' = 10\beta$.

To see why we only need to consider the perfectly unbalanced case, [11] proved the following two Lemmas.

Lemma 3.3.1. *If $|1(x) - 0(y)|\delta \le \delta\alpha$, then*

$$\mathsf{BestMatch}(x,y) \ge (1/2 + \delta/2)\, \mathit{LCS}(x,y).$$

Lemma 3.3.2. *Let $\beta', \gamma > 0$ be sufficiently small constants. If $0(x) \in [1/2 - \beta', 1/2 + \beta']$, then*

$$\max\{\mathsf{BestMatch}(x,y), \mathsf{ApxED}(x,y)\} \le (1/2 + \gamma)\, \mathit{LCS}(x,y).$$

If the two input string are not perfectly unbalanced, we can compute $\mathsf{BestMatch}(x,y)$ and ApxED to get a $1/2 + \varepsilon$ approximation fo $\mathsf{LCS}(x,y)$ for some small constant $\varepsilon > 0$.

Given two strings in the perfectly unbalanced case, without loss of generality, we assume $1(y) = \alpha$. The algorithm first both strings x, y into three parts such that $x = L_x \circ M_x \circ R_x$ and $y = L_y \circ M_y \circ R_y$ where $|L_x| = |R_x| = |L_y| = |R_y| = \alpha n$. Then, the inputs are divided into six cases according to the first order statistics (number of 0's and 1's) of L_x, R_x, L_y, R_y. For each case, we can use the four ingredient algorithms to get a $(1/2 + \varepsilon)$ approximation of $\mathsf{LCS}(x, y)$ for some small constant ε. We refer readers to [11] for the pseudocode and analysis of the algorithm. We omit the details here.

We now prove Theorem 3.3.

Proof of Theorem 3.3. As usual, we assume x is the online string and y is the offline string. If the two input strings are not perfectly unbalanced, we can compute $\mathsf{BestMatch}(x, y)$ in $O(\log n)$ space in the asymmetric streaming model since we only need to compare the first order statistics of x and y. Also, for any constant δ we can compute a constant approximation (dependent on δ) of $\mathsf{ED}(x, y)$ using $\tilde{O}(n^\delta)$ space by Lemma 3.2.8. Thus, we only need to consider the case where x and y are perfectly balanced.

Notice that the algorithm from [11] needs to compute Match, $\mathsf{BestMatch}$, Greedy, ApxED with input strings chosen from x, L_x, $L_x \circ M_x$, R_x, $M_x \circ R_x$ and y, L_y, $L_y \circ M_y$, R_y, $M_y \circ R_y$.

If we know the number of 1's and 0's in L_x, M_x, and R_x, then we can compute Match and $\mathsf{BestMatch}$ with any pair of input strings from x, L_x, $L_x \circ M_x$, R_x, $M_x \circ R_x$ and y, L_y, $L_y \circ M_y$, R_y, $M_y \circ R_y$..

For ApxED, according to the algorithm in [11], we only need to compute $\mathsf{ApxED}(x, y)$, $\mathsf{ApxED}(L_x, L_y)$, and $\mathsf{ApxED}(R_x, R_y)$. For any constant $\delta > 0$, we can get a constant approximation (dependent on δ) of edit distance with $\tilde{O}(n^\delta)$ space in the asymmetric streaming model by Lemma 3.2.8.

For Greedy, there are two cases. For the first case, the online string x is divided into two parts and the input strings are x^1, x^2, y where $x = x^1 \circ x^2$. Notice that x^1 can only be L_x or $L_x \circ M_x$. In this case, we only need to remember $1(L_x)$, $1(M_x)$, and $1(R_x)$. Since the length of L_x, M_x, and R_x are all fixed. We know $0(L_x) = |L_x| - 1(L_x)$, and similar for M_x and R_x. We know for $l \in [n]$

$$\begin{aligned}
&\mathsf{BestMatch}(y[1:l], x^1) + \mathsf{BestMatch}(y[l+1:n], x^2)\\
=& \max_{\sigma \in \{0,1\}} \mathsf{Match}(y[1:l], x^1, \sigma) + \max_{\sigma \in \{0,1\}} \mathsf{Match}(y[l+1:n], x^2, \sigma)\\
=& \max_{\sigma \in \{0,1\}} \Big(\min\{\sigma(y[1:l]), \sigma(x^1)\}\Big) + \max_{\sigma \in \{0,1\}} \Big(\min\{\sigma(y[l+1:n]), \sigma(x^2)\}\Big)
\end{aligned}$$

Given $\alpha = 1(y)$, we know $1(y[l+1:n]) = \alpha - 1(y[1:l])$, $0(y[1:l]) = l/n - 0(y[1:l])$, and $0(y[l+1:n]) = (n-l)/n - 0(y[l+1:n])$. Thus we only need to read y from left to right once and remember the index l that maximizes $\mathsf{BestMatch}(y[1:l], x^1) + \mathsf{BestMatch}(y[l+1:n], x^2)$.

For the case when the input strings are y^1, y^2, x, if we know $0(x)$, similarly, we can compute $\mathsf{Greedy}(y^1, y^1, x)$ by reading x from left to right once with $O(\log n)$ bits of space. Here, $0(x)$ is not known to us before computation. However, in the perfectly unbalanced case, we assume $|1(y) - 0(x)| < \delta$ is a sufficiently small constant. We can simply assume $0(x) = 1(y) = \alpha$ and run $\mathsf{BestMatch}(y^1, y^2, x)$ in the asymmetric streaming model. This will add an error of at most δ. The algorithm still outputs a $(1/2 + \varepsilon)$ approximation of $\mathsf{LCS}(x, y)$ for some small constant $\varepsilon > 0$.

$\square$

3.4 Open Problems

It would be interesting to see either algorithms with improved performance or stronger space lower bounds. For example, can we design a $1+\varepsilon$ approximation algorithm for ED or LCS in the asymmetric streaming model with $o(\sqrt{n})$ space? We present several space lower bounds in Chapter 4. However, there are still gaps between our lower bounds and upper bounds in some parameter regimes (more discussion can be find in Section 4.5). Also, we note all algorithms presented in this chapter are deterministic. It would be interesting to see if randomness can be used to design better algorithms.

Chapter 4

Asymmetric Streaming Model: Lower Bounds

4.1 Introduction

In this chapter, we discuss several space lower bounds for computing or approximating ED and LCS in the asymmetric streaming model. Some bounds also extend to LIS and LNS.

4.1.1 Main Results

To simplify notation we always use $1+\varepsilon$ approximation for some $\varepsilon > 0$, i.e., outputting an λ with $\mathsf{OPT} \leq \lambda \leq (1+\varepsilon)\mathsf{OPT}$, where OPT is either $\mathsf{ED}(x, y)$ or $\mathsf{LCS}(x, y)$. We note that for LCS, this is equivalent to a $1/(1+\varepsilon)$ approximation in the standard notation.

Previously, there are no explicitly stated space lower bounds in this model, although as we will discuss later, some lower bounds about LCS can be inferred from the lower bounds for *longest increasing subsequence* LIS in [37, 42, 43]. As our first contribution, we prove strong lower bounds for ED in the asymmetric streaming model.

Theorem 4.1. *There is a constant $c > 1$ such that for any $k, n \in \mathbb{N}$ with $n \geq ck$, given an alphabet Σ, any R-pass randomized algorithm in the asymmetric streaming model that decides if $\mathsf{ED}(x, y) \geq k$ for two strings $x, y \in \Sigma^n$ with success probability*

$\geq 2/3$ *must use space* $\Omega(\mathsf{min}(k, |\Sigma|)/R)$.

This theorem implies the following corollary.

Corollary 4.1.1. *Given an alphabet* Σ*, the following space lower bounds hold for any constant pass randomized algorithm with success probability* $\geq 2/3$ *in the asymmetric streaming model.*

1. $\Omega(n)$ *for computing* $\mathsf{ED}(x, y)$ *of two strings* $x, y \in \Sigma^n$ *if* $|\Sigma| \geq n$.

2. $\Omega(\frac{1}{\varepsilon})$ *for* $1+\varepsilon$ *approximation of* $\mathsf{ED}(x, y)$ *for two strings* $x, y \in \Sigma^n$ *if* $|\Sigma| \geq 1/\varepsilon$.

Our theorems thus provide a justification for the study of *approximating* ED in the asymmetric streaming model. Furthermore, we note that previously, unconditional lower bounds for ED in various computational models are either weak, or almost identical to the bounds for Hamming distance. For example, a simple reduction from the equality function implies the deterministic two party communication complexity (and hence also the space lower bound in the standard streaming model) for computing or even approximating ED is $\Omega(n)$.[1] However the same bound holds for Hamming distance. Thus it has been an intriguing question to prove a rigorous, unconditional separation of the complexity of ED and Hamming distance. To the best of our knowledge the only previous example achieving this is the work of [121] and [122], which showed that the randomized two party communication complexity of achieving a $1+\varepsilon$ approximation of ED is $\Omega(\frac{\log n}{(1+\varepsilon)\log\log n})$, while the same problem for Hamming distance has an upper bound of $O(\frac{1}{\varepsilon^2})$. Thus if ε is a constant, this provides a separation of $\Omega(\frac{\log n}{\log\log n})$ vs. a constant. However, this result also has some disadvantages: (1) It only works in the randomized setting; (2) The separation becomes obsolete when ε is small, e.g., $\varepsilon = 1/\sqrt{\log n}$; and (3) The lower bound for ED is still weak and thus it

[1]We include this bound in Chapter B for completeness, as we cannot find any explicit statement in the literature.

does not apply to the streaming setting, as there even recoding the index needs space $\log n$.

Our result from Corollary 4.1.1, on the other hand, complements the above result in the aforementioned aspects by providing another strong separation of ED and Hamming distance. Note that even exact computation of the Hamming distance between x and y is easy in the asymmetric streaming model with one pass and space $O(\log n)$. Thus our result provides an *exponential* gap between edit distance and Hamming distance, in terms of the space complexity in the asymmetric streaming model (and also the communication model since our proof uses communication complexity), even for deterministic exact computation.

Next we turn to LCS, which can be viewed as a generalization of LIS. For example, if the alphabet $\Sigma = [n]$, then we can fix the string y to be the concatenation from 1 to n, and it's easy to see that $\mathsf{LCS}(x, y) = \mathsf{LIS}(x)$. Therefore, the lower bound of computing LIS for randomized streaming in [37] with $|\Sigma| \geq n$ also implies a similar bound for LCS in the asymmetric streaming model. However, the bound in [37] does not apply to the harder case where x is a *permutation* of y, and their lower bound where $|\Sigma| < n$ is actually for *longest non-decreasing subsequence*, which does not give a similar bound for LCS in the asymmetric streaming model. [2] Therefore, we first prove a strong lower bound for LCS in general.

Theorem 4.2. *There is a constant $c > 1$ such that for any $k, n \in \mathbb{N}$ with $n \geq ck$, given an alphabet Σ, any R-pass randomized algorithm in the asymmetric streaming model that decides if $\mathsf{LCS}(x, y) \geq k$ for two strings $x, y \in \Sigma^n$ with success probability $\geq 2/3$ must use space $\Omega\Big(\mathsf{min}(k, |\Sigma|)/R\Big)$. Moreover, this holds even if x is a permutation of y when $|\Sigma| \geq n$ or $|\Sigma| \leq k$.*

Similar to the case of ED, this theorem also implies the following corollary.

[2] One can get a similar reduction to LCS, but now y needs to be the sorted version of x, which gives additional information about x in the asymmetric streaming model since we have random access to y.

Corollary 4.1.2. *Given an alphabet Σ, the following space lower bounds hold for any constant pass randomized algorithm with success probability $\geq 2/3$ in the asymmetric streaming model.*

1. *$\Omega(n)$ for computing $\mathsf{LCS}(x, y)$ of two strings $x, y \in \Sigma^n$ if $|\Sigma| \geq n$.*

2. *$\Omega(\frac{1}{\varepsilon})$ for $1 + \varepsilon$ approximation of $\mathsf{LCS}(x, y)$ for two strings $x, y \in \Sigma^n$ if $|\Sigma| \geq 1/\varepsilon$.*

We then consider *deterministic* approximation of LCS. Here, the work of [42, 43] gives a lower bound of $\Omega\left(\frac{1}{R}\sqrt{\frac{n}{\varepsilon}}\log\left(\frac{|\Sigma|}{\varepsilon n}\right)\right)$ for any R pass streaming algorithm achieving a $1 + \varepsilon$ approximation of LIS, which also implies a lower bound of $\Omega\left(\frac{1}{R}\sqrt{\frac{n}{\varepsilon}}\log\left(\frac{1}{\varepsilon}\right)\right)$ for asymmetric streaming LCS when $|\Sigma| \geq n$. These bounds match the upper bound in [14] for LIS and LNS, and our algorithm in Chapter 3 for LCS. However, a major drawback of this bound is that it gives nothing when $|\Sigma|$ is small (e.g., $|\Sigma| \leq \varepsilon n$). For even smaller alphabet size, the bound does not even give anything for exact computation. For example, in the case of a binary alphabet, we know that $\mathsf{LIS}(x) \leq 2$ and thus taking $\varepsilon = 1/2$ corresponds to exact computation. Yet the bound gives a negative number.

This is somewhat disappointing as in most applications of ED and LCS, the alphabet size is actually a fixed constant. These include for example the English language and the human DNA sequence (where the alphabet size is 4 for the 4 bases). Therefore, we focus on the case where the alphabet size is small, and we have the following theorem.

Theorem 4.3. *Given an alphabet Σ, for any $\varepsilon > 0$ where $\frac{|\Sigma|^2}{\varepsilon} = O(n)$, any R-pass deterministic algorithm in the asymmetric streaming model that computes a $1 + \varepsilon$ approximation of $\mathsf{LCS}(x, y)$ for two strings $x, y \in \Sigma^n$ must use space $\Omega\left(\frac{|\Sigma|}{\varepsilon}/R\right)$.*

Thus, even for a binary alphabet, achieving $1 + \varepsilon$ approximation for small ε (e.g., $\varepsilon = 1/n$ which corresponds to exact computation) can take space as large as $\Omega(n)$ for any constant pass algorithm. Further note that by taking $|\Sigma| = \sqrt{\varepsilon n}$, we recover the $\Omega\left(\frac{\sqrt{n}}{\varepsilon}/R\right)$ bound with a much smaller alphabet.

Finally, we turn to LIS and longest non-decreasing subsequence (LNS), as well as a natural generalization of LIS and LNS which we call *longest non-decreasing subsequence with threshold* (LNST). Given a string $x \in \Sigma^n$ and a threshold $t \leq n$, $\mathsf{LNST}(x,t)$ denotes the length of the longest non-decreasing subsequence in x such that each symbol appears at most t times. It is easy to see that the case of $t = 1$ corresponds to LIS and the case of $t = n$ corresponds to LNS. Thus LNST is indeed a generalization of both LIS and LNS. It is also a special case of LCS when $|\Sigma|t \leq n$ as we can take y to be the concatenation of t copies of each symbol, in the ascending order (and possibly padding some symbols not in x). How hard is LNST? We note that in the case of $t = 1$ (LIS) and $t = n$ (LNS) a simple dynamic programming can solve the problem in one pass with space $O(|\Sigma| \log n)$, and $1 + \varepsilon$ approximation can be achieved in one pass with space $\tilde{O}(\sqrt{\frac{n}{\varepsilon}})$ by [14]. Thus one can ask what is the situation for other t. Again we focus on the case of a small alphabet and have the following theorem.

Theorem 4.4. *Given an alphabet Σ, for deterministic $(1 + \varepsilon)$ approximation of $\mathsf{LNST}(x,t)$ for a string $x \in \Sigma^n$ in the streaming model with R passes, we have the following space lower bounds:*

1. *$\Omega(\min(\sqrt{n}, |\Sigma|)/R)$ for any constant t (this includes LIS), when ε is any constant.*
2. *$\Omega(|\Sigma| \log(1/\varepsilon)/R)$ for $t \geq n/|\Sigma|$ (this includes LNS), when $|\Sigma|^2/\varepsilon = O(n)$.*
3. *$\Omega\left(\frac{\sqrt{|\Sigma|}}{\varepsilon}/R\right)$ for $t = \Theta(1/\varepsilon)$, when $|\Sigma|/\varepsilon = O(n)$.*

Thus, case 1 and 2 show that even for any constant approximation, any constant pass streaming algorithm for LIS and LNS needs space $\Omega(|\Sigma|)$ when $|\Sigma| \leq \sqrt{n}$, matching the $O(|\Sigma| \log n)$ upper bound up to a logarithmic factor. Taking $\varepsilon = 1/\sqrt[3]{n}$ and $|\Sigma| \leq \sqrt[3]{n}$ for example, we further get a lower bound of $\Omega(|\Sigma| \log n)$ for approximating LNS using any constant pass streaming algorithm. This matches the $O(|\Sigma| \log n)$ upper bound. These results complement the bounds in [14, 42, 43] for the important case of small alphabet, and together they provide an almost complete picture for LIS and LNS. Case 3 shows that for certain choices of t and ε, the space we need for LNST can be significantly larger than those for LIS and LNS. It is an intriguing question to completely characterize the behavior of LNST for all regimes of parameters.

Finally, we provide a simple algorithm that can use much smaller space for certain regimes of parameters.

Theorem 4.5. *Given an alphabet Σ with $|\Sigma| = r$. For any $\varepsilon > 0$ and $t \geq 1$, there is a one-pass streaming algorithm that computes a $(1+\varepsilon)$ approximation of $\mathsf{LNST}(x,t)$ for any $x \in \Sigma^n$ with $\tilde{O}\Big(\big(\min(t, r/\varepsilon)+1\big)^r\Big)$ space.*

4.1.2 Overview of Techniques

Our lower bounds use the general framework of communication complexity. To limit the power of random access to the string y, we always fix y to be a specific string, and consider different strings x. In turn, we divide x into several blocks and consider the two party/multi party communication complexity of $\mathsf{ED}(x,y)$ or $\mathsf{LCS}(x,y)$, where each party holds one block of x. However, we need to develop several new techniques to handle edit distance and small alphabets.

Edit Distance. We start with edit distance. One difficulty here is to handle substitutions, as with substitutions edit distance becomes similar to Hamming distance, and this is exactly one of the reasons why strong complexity results separating edit distance and Hamming distance are rare. Indeed, if we define $\mathsf{ED}(x,y)$ to be the smallest number of insertions and deletions (without substitutions) to transform x into y, then $\mathsf{ED}(x,y) = 2n - 2\,\mathsf{LCS}(x,y)$ and thus a lower bound for exactly computing LCS (e.g., those implied from [42, 43]) would translate directly into the same bound for exactly computing ED. On the other hand, with substitutions things become more complicated: if $\mathsf{LCS}(x,y)$ is small (e.g., $\mathsf{LCS}(x,y) \leq n/2$) then in many cases (such as examples obtained by reducing from [42, 43]) the best option to transform x into y is just replacing each symbol in x by the corresponding symbol in y if they are different, which makes $\mathsf{ED}(x, y)$ exactly the same as their Hamming distance.

To get around this, we need to ensure that $\mathsf{LCS}(x, y)$ is large. We demonstrate our ideas by first describing an $\Omega(n)$ lower bound for the deterministic two party communication complexity of $\mathsf{ED}(x, y)$, using a reduction from the equality function which is well known to have an $\Omega(n)$ communication complexity bound. Towards this, fix $\Sigma = [3n] \cup \{a\}$ where a is a special symbol, and fix $y = 1 \circ 2 \circ \cdots \circ 3n$. We divide x into two parts $x = (x_1, x_2)$ such that x_1 is obtained from the string $(1, 2, 4, 5, \cdots, 3i-2, 3i-1, \cdots, 3n-2, 3n-1)$ by replacing some symbols of the form $3j-1$ by a, while x_2 is obtained from the string $(2, 3, 5, 6, \cdots, 3i-1, 3i, \cdots, 3n-1, 3n)$ by replacing some symbols of the form $3j-1$ by a. Note that the way we choose (x_1, x_2) ensures that $\mathsf{LCS}(x, y) \geq 2n$ before replacing any symbol by a.

Intuitively, we want to argue that the best way to transform x into y, is to delete a substring at the end of x_1 and a substring at the beginning of x_2, so that the resulted string becomes an increasing subsequence as long as possible. Then, we insert symbols into this string to make it match y except for those a symbols. Finally, we replace the a symbols by substitutions. If this is true then we can finish the argument as follows. Let $T_1, T_2 \subset [n]$ be two subsets with size $t = \Omega(n)$, where for any $i \in \{1, 2\}$, all symbols of the form $3j-1$ in x_i with $j \in T_i$ are replaced by a. Now if $T_1 = T_2$ then it doesn't matter where we choose to delete the substrings in x_1 and x_2, the number of edit operations is always $3n-2+t$ by a direct calculation. On the other hand if $T_1 \neq T_2$ and assume for simplicity that the smallest element they differ is an element in T_2, then there is a way to save one substitution, and the the number of edit operations becomes $3n-3+t$.

The key part is now proving our intuition. For this, we consider all possible $r \in [3n]$ such that x_1 is transformed into $y[1:r]$ and x_2 is transformed into $y[r+1:3n]$, and compute the two edit distances respectively. To analyze the edit distance, we first show by a greedy argument that without loss of generality, we can assume that we apply deletions first, followed by insertions, and substitutions at last. This reduces the edit distance problem to the following problem: for a fixed number of deletions and insertions, what is the best way to minimize the Hamming distance (or maximize the number of agreements of symbols at the same indices) in the end. Now we break the analysis of $\mathsf{ED}(x_1, y[1:r])$ into two cases. Case 1 is where the number of deletions (say d_d) is large. In this case, the number of insertions (say d_i) must also be large, and we argue that the number of agreements is at most $\mathsf{LCS}(x_1, y[1:r]) + d_i$. Case 2 is where d_d is small. In this case, d_i must also be small. Now we crucially use the structure of x_1 and y, and argue that symbols in x_1 larger than $3d_i$ (or original index beyond $2d_i$) are guaranteed to be out of agreement. Thus the number of agreements is at most $\mathsf{LCS}(x_1[1:2d_i], y[1:r]) + d_i$. In each case combining the bounds gives us a lower bound on the total number of operations. The situation for x_2 and $y[r+1:3n]$ is completely symmetric and this proves our intuition.

In the above construction, x and y have different lengths ($|x| = 4n$ while $|y| = 3n$). We can fix this by adding a long enough string z with distinct symbols than those in $\{x, y\}$ to the end of both x and y, and then add n symbols of a at the end of z for y. We argue that the best way to do the transformation is to transform x into y, and then insert n symbols of a. To show this, we first argue that at least one symbol in z must be kept, for otherwise the number of operations is already larger than the previous transformation. Then, using a greedy argument we show that the entire z must be kept, and thus the natural transformation is the optimal.

To extend the bound to randomized algorithms, we modify the above construction and reduce from *Set Disjointness* (DIS), which is known to have randomized communication complexity $\Omega(n)$. Given two strings $\alpha, \beta \in \{0,1\}^n$ representing the characteristic vectors of two sets $A, B \subseteq [n]$, $\mathsf{DIS}(\alpha, \beta) = 0$ if and only if $A \cap B \neq \emptyset$, or equivalently, $\exists j \in [n], \alpha_j = \beta_j = 1$. For the reduction, we first create two new strings $\alpha', \beta' \in \{0,1\}^{2n}$ which are "balanced" versions of α, β. Formally, $\forall j \in [n], \alpha'_{2j-1} = \alpha_j$ and $\alpha'_{2j} = 1 - \alpha_j$. We create β' slightly differently, i.e., $\forall j \in [n], \beta'_{2j-1} = 1 - \beta_j$ and $\beta'_{2j} = \beta_j$. Now both α' and β' have n 1's, we can use them as the characteristic vectors of the two sets T_1, T_2 in the previous construction. A similar argument now leads to the bound for randomized algorithms.

Longest Common Subsequence. Our lower bounds for randomized algorithms computing LCS exactly are obtained by a similar and simpler reduction from DIS: we still fix y to be an increasing sequence of length $8n$ and divide y evenly into $4n$ blocks of constant size. Now x_1 consists of the blocks with an odd index, while x_2 consists of the blocks with an even index. Thus x is a permutation of y. Next, from $\alpha, \beta \in \{0,1\}^n$ we create $\alpha', \beta' \in \{0,1\}^{2n}$ in a slightly different way and use α', β' to modify the $2n$ blocks in x_1 and x_2 respectively. If a bit is 1 then we arrange the corresponding block in the increasing order, otherwise we arrange the corresponding block in the decreasing order. A similar argument as before now gives the desired $\Omega(n)$ bound. We note that [37] has similar results for LIS by reducing from DIS. However, our reduction and analysis are different from theirs. Thus we can handle LCS, and even the harder case where x is a permutation of y.

We now turn to LCS over a small alphabet. To illustrate our ideas, let's first consider $\Sigma = \{0,1\}$ and choose $y = 0^{n/2}1^{n/2}$. It is easy to see that $\mathsf{LCS}(x,y) = \mathsf{LNST}(x, n/2)$. We now represent each string $x \in \{0,1\}^n$ as follows: at any index $i \in [n] \cup \{0\}$, we record a pair (p,q) where $p = \mathsf{min}(\text{the number of 0's in } x[1:i], n/2)$ and $q = \mathsf{min}(\text{the number of 1's in } x[i+1:n], n/2)$. Thus, if we read x from left to right, then upon reading a 0, p may increase by 1 and q does not change; while upon reading a 1, p does not change and q may decrease by 1. Hence if we use the horizontal axis to stand for p and the vertical axis to stand for q, then these points (p,q) form a polygonal chain. We call $p+q$ the *value* at point (p,q) and it is easy to see that $\mathsf{LCS}(x,y)$ must be the value of an endpoint of some chain segment.

Using the above representation, we now fix $\Sigma = \{0,1,2\}$ and choose $y = 0^{n/3}1^{n/3}2^{n/3}$, so $\mathsf{LCS}(x,y) = \mathsf{LNST}(x, n/3)$. We let $x = (x_1, x_2)$ such that $x_1 \in \{0,1\}^{n/2}$ and $x_2 \in \{1,2\}^{n/2}$. Since any common subsequence between x and y must be of the form $0^a1^b2^c$ it suffices to consider common subsequence between x_1 and $0^{n/3}1^{n/3}$, and that between x_2 and $1^{n/3}2^{n/3}$, and combine them together. Towards that, we impose the following properties on x_1, x_2: (1) The number of 0's, 1's, and 2's in each string is at most $n/3$; (2) In the polygonal chain representation of each string, the values of the endpoints strictly increase when the number of 1's increases; and (3) For any endpoint in x_1 where the number of 1's is some r, there is a corresponding endpoint in x_2 where the number of 1's is $n/3 - r$, and the values of these two endpoints sum up to a fixed number $t = \Omega(n)$. Note that property (2) implies that $\mathsf{LCS}(x,y)$ must be the sum of the values of an endpoint in x_1 where the number of 1's is some r, and an endpoint in x_2 where the number of 1's is $n/3 - r$, while property (3) implies that for any string x_1, there is a unique corresponding string x_2, and $\mathsf{LCS}(x,y) = t$ (regardless of the choice of r).

We show that under these properties, all possible strings $x = (x_1, x_2)$ form a set S with $|S| = 2^{\Omega(n)}$, and this set gives a *fooling set* for the two party communication problem of computing $\mathsf{LCS}(x,y)$. Indeed, for any $x = (x_1, x_2) \in S$, we have $\mathsf{LCS}(x,y) = t$. On the other hand, for any $(x_1, x_2) \neq (x'_1, x'_2) \in S$, the values must differ at some point for x_1 and x'_1. Hence by switching, either (x_1, x'_2) or (x'_1, x_2) will have a LCS with y that has length at least $t+1$. Standard arguments now imply an $\Omega(n)$ communication complexity lower bound. A more careful analysis shows that we can even replace the symbol 2 by 0, thus resulting in a binary alphabet.

The above argument can be easily modified to give a $\Omega(1/\varepsilon)$ bound for $1+\varepsilon$ approximation of LCS when $\varepsilon < 1$, by taking the string length to be some $n' = \Theta(1/\varepsilon)$.

To get a better bound, we combine our technique with the technique in [43] and consider the following direct sum problem: we create r copies of strings $\{x^i, i \in [r]\}$ and $\{y^i, i \in [r]\}$ where each copy uses distinct alphabets with size 2. Assume for x^i and y^i the alphabet is $\{a_i, b_i\}$, now x^i again consists of r copies of $(x^i_{j1}, x^i_{j2}), j \in [r]$, where each $x^i_{j\ell} \in \{a_i, b_i\}^{n'/2}$ for $\ell \in [2]$; while y^i consists of r copies $y^i_j = a_i^{n'/3} b_i^{n'/3} a_i^{n'/3}, j \in [r]$. The direct sum problem is to decide between the following two cases for some $t = \Omega(n')$: (1) $\exists i$ such that there are $\Omega(r)$ copies (x^i_{j1}, x^i_{j2}) in x^i with $\mathsf{LCS}((x^i_{j1} \circ x^i_{j2}), y^i_j) \geq t+1$, and (2) $\forall i$ and $\forall j$, $\mathsf{LCS}((x^i_{j1} \circ x^i_{j2}), y^i_j) \leq t$. We do this by arranging the x^i's row by row into an $r \times 2r$ matrix (each entry is a length $n'/2$ string) and letting x be the concatenation of the *columns*. We call these strings the *contents* of the matrix, and let y be the concatenation of the y^i's. Now intuitively, case (1) and case (2) correspond to deciding whether $\mathsf{LCS}(x, y) \geq 2rt + \Omega(r)$ or $\mathsf{LCS}(x, y) \leq 2rt$, which implies a $1 + \Omega(1/t) = 1 + \varepsilon$ approximation. The lower bound follows by analyzing the $2r$-party communication complexity of this problem, where each party holds a column of the matrix.

However, unlike the constructions in [42, 43] which are relatively easy to analyze because all symbols in x (respectively y) are *distinct*, the repeated symbols in our construction make the analysis of LCS much more complicated (we can also use distinct symbols but that will only give us a bound of $\frac{\sqrt{|\Sigma|}}{\varepsilon}$ instead of $\frac{|\Sigma|}{\varepsilon}$). To ensure that the LCS is to match each (x^i_{j1}, x^i_{j2}) to the corresponding y^i_j, we use another r symbols $\{c_i, i \in [r]\}$ and add *buffers* of large size (e.g., size n') between adjacent copies of (x^i_{j1}, x^i_{j2}). We do the same thing for y^i_j correspondingly. Moreover, it turns out we need to arrange the buffers carefully to avoid unwanted issues: in each row x^i, between each copy of (x^i_{j1}, x^i_{j2}) we use a buffer of new symbol. Thus the buffers added to each row x^i are $c_1^{n'}, c_2^{n'}, \cdots, c_r^{n'}$ sequentially and this is the same for every row. That is, in each row the contents use the same alphabet $\{a_i, b_i\}$ but the buffers use different alphabets $\{c_i, i \in [r]\}$. Now we have a $r \times 3r$ matrix and we again let x be the concatenation of the *columns* while let y be the concatenation of the y^i's. Note that we are using an alphabet of size $|\Sigma| = 3r$. We use a careful analysis to argue that case (1) and case (2) now correspond to deciding whether $\mathsf{LCS}(x, y) \geq 2rn' + rt + \Omega(r)$ or $\mathsf{LCS}(x, y) \leq 2rn' + rt$, which implies a $1 + \varepsilon$ approximation. The lower bound follows by analyzing the $3r$-party communication complexity of this problem, and we show a lower bound of $\Omega(r/\varepsilon) = \Omega(|\Sigma|/\varepsilon)$ by generalizing our previous fooling set construction to the multi-party case, where we use a good error correcting code to create the $\Omega(r)$ gap.

The above technique works for $\varepsilon < 1$. For the case of $\varepsilon \geq 1$ our bound for LCS can be derived directly from our bound for LIS, which we describe next.

Longest Increasing/Non-decreasing Subsequence. Our $\Omega(|\Sigma|)$ lower bound over small alphabet is achieved by modifying the construction in [43] and providing a better analysis. Similar as before, we consider a matrix $B \in \{0, 1\}^{\frac{r}{c} \times r}$ where c is a large constant and $r = |\Sigma|$. We now consider the r-party communication problem where each party holds one column of B, and the problem is to decide between the following two cases for a large enough constant l: (1) for each row in B, there are at least l 0's between any two 1's, and (2) there exists a row in B which has more than αr 1's, where $\alpha \in (1/2, 1)$ is a constant. We can use a similar argument as in [43] to show that the total communication complexity of this problem is $\Omega(r^2)$ and hence at least one party needs $\Omega(r)$. The difference is that [43] sets $l = 1$ while we need to pick l to be a larger constant to handle the case $\varepsilon \geq 1$. For this we use the Lovász Local Lemma with a probabilistic argument to show the existence of a large fooling set. To reduce to LIS, we define another matrix $\tilde{B}$ such that $\tilde{B}_{i,j} = (i-1)\frac{r}{c} + j$ if $B_{i,j} = 1$ and $\tilde{B}_{i,j} = 0$ otherwise. Now let x be the concatenation of all columns of $\tilde{B}$. We show that case (2) implies $\mathsf{LIS}(x) \geq \alpha r$ and case (1) implies $\mathsf{LIS}(x) \leq (1/c + 1/l)r$. This implies a $1 + \varepsilon$ approximation for any constant $\varepsilon > 0$ by setting c and l appropriately.

The construction is slightly different for LNS. This is because if we keep the 0's in $\tilde{B}$, they will already form a very long non-decreasing subsequence and we will not get any gap. Thus, we now let the matrix B have size $r \times cr$ where c can be any constant. We replace all 0's in column i with a symbol b_i for $i \in [cr]$, such that $b_1 > b_2 > \cdots > b_{cr}$. Similarly we replace all 1's in row j with a symbol a_j for $j \in [r]$, such that $a_1 < a_2 < \cdots < a_r$. Also, we let $a_1 > b_1$. We can show that the two cases now correspond to $\mathsf{LNS}(x) > \alpha cr$ and $\mathsf{LNS}(x) \leq (2 + c/l)r$.

We further prove an $\Omega(|\Sigma| \log(1/\varepsilon))$ lower bound for $1+\varepsilon$ approximation of LNS when $\varepsilon < 1$. This is similar to our previous construction for LCS, except we don't need buffers here, and we only need to record the number of some symbols. More specifically, let $l = \Theta(1/\varepsilon)$ and S be the set of all strings $x = (x_1, x_2)$ over alphabet $\{a, b\}$ with length $2l$ such that $x_1 = a^{\frac{3}{4}l+t}b^{\frac{1}{4}l-t}$ and $x_2 = a^{\frac{3}{4}l-t}b^{\frac{1}{4}l+t}$ for any $t \in [\frac{l}{4}]$. Thus S has size $\frac{l}{4} = \Omega(1/\varepsilon)$ and $\forall x \in S$, the number of a's in x is exactly $\frac{3}{2}l$. Further, for any $(x_1, x_2) \neq (x'_1, x'_2) \in S$, either (x_1, x'_2) or (x'_1, x_2) has more than $\frac{3}{2}l$ a's. We now consider the $r \times 2r$ matrix where each row i consists of $\{(x^i_{j1}, x^i_{j2}), j \in [r]\}$ such that each $x^i_{j\ell}$ has length l for $\ell \in [2]$, and for the same row i all $\{(x^i_{j1}, x^i_{j2})\}$ use the same alphabet $\{a_i, b_i\}$ while for different rows the alphabets are disjoint. To make sure the LNS of the concatenation of the columns is roughly the sum of the number of a_i's, we require that $b_r < b_{r-1} < \cdots < b_1 < a_1 < a_2 < \cdots < a_r$. Now we analyze the $2r$ party communication problem of deciding whether the concatenation of the columns has $\mathsf{LNS} \geq crl + \Omega(r)$ or $\mathsf{LNS} \leq crl$ for some constant c, which implies a $1+\varepsilon$ approximation. The lower bound is again achieved by generalizing the set S to a fooling set for the $2r$ party communication problem using an error correcting code based approach.

In Theorem 4.4, we give three lower bounds for LNST. The first two lower bounds are adapted from our lower bounds for LIS and LNS, while the last lower bound is adapted from our lower bound for LCS by ensuring all symbols in different rows or columns of the matrix there are different.

4.1.3 Preliminaries

In the proofs, we sometime consider the matching between $x, y \in \Sigma^n$. By a matching, we mean a function $m : [n] \to [n] \cup \emptyset$ such that if $m(i) \neq \emptyset$, we have $x_i = y_{m(i)}$. We require the matching to be non-crossing. That is, for $i < j$, if $m(i)$ and $m(j)$ are both not $\emptyset$, we have $m(i) < m(j)$. The size of a matching is the number of $i \in [n]$ such that $m(i) \neq \emptyset$. We say a matching is a best matching if it achieves the maximum size. Each matching between x and y corresponds to a common subsequence. Thus, the size of a best matching between x and y is equal to $\mathsf{LCS}(x, y)$.

We use the following form of Lovász Local Lemma.

Lemma 4.1.1 (Lovász Local Lemma)**.** *Let $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ be a finite set of events. For $A \in \mathcal{A}$, let $\Gamma(A)$ denote the neighbours of A in the dependency graph (In the dependency graph, mutually independent events are not adjacent). If there exist an assignment of reals $x : \mathcal{A} \to [0, 1)$ to the events such that*

$$\forall\ A \in \mathcal{A},\ \Pr(A) \leq x(A) \prod_{A' \in \Gamma(A)} (1 - x(A')).$$

Then, for the probability that none of the events in $\mathcal{A}$ happens, we have

$$\Pr(\overline{A_1} \wedge \overline{A_2} \wedge \cdots \wedge \overline{A_{t-l}}) \geq \prod_{i=1}^{n} (1 - x(A_i)).$$

In the **Set Disjointness** problem, we consider a two party game. Each party holds a binary string of length n, say x and y. And the goal is to compute the function $\mathsf{DIS}(x, y)$ as defined below

$$\mathsf{DIS}(x, y) = \begin{cases} 0, & \text{if } \exists\ i, \text{ s.t. } x_i = y_i \\ 1, & \text{otherwise} \end{cases}$$

We define $R^{1/3}(f)$ as the minimum number of bits required to be sent between two parties in any randomized multi-round communication protocol with 2-sided error at most 1/3. The following is a well-known result.

Lemma 4.1.2 ([123], [124])**.** *$R^{1/3}(\mathsf{DIS}) = \Omega(n)$.*

We will consider the one-way t-party communication model where t players $P_1, P_2, \ldots, P_t$ each holds input $x_1, x_2, \ldots, x_t$ respectively. The goal is to compute the function $f(x_1, x_2, \ldots, x_t)$. In the one-way communication model, each player speaks in turn and player P_i can only send message to player P_{i+1}. We sometimes consider multiple round of communication. In an R round protocol, during round $r \leq R$, each player speaks in turn P_i sends message to P_{i+1}. At the end of round $r < R$, player P_t sends a message to P_1. At the end of round R, player P_t must output the answer of the protocol. We note that our lower bound also hold for a stronger *blackboard* model. In this model, the players can write messages (in the order of $1, \ldots, t$) on a blackboard that is visible to all other players.

We define the *total communication complexity* of f in the t-party one-way communication model, denoted by $CC_t^{tot}(f)$, as the minimum number of bits required to be sent by the players in every deterministic communication protocol that always outputs a correct answer. The total communication complexity in the blackboard model is the total length of the messages written on the blackboard by the players.

For deterministic protocol P that always outputs the correct answer, we let $M(P)$ be the maximum number of bits required to be sent by some player in protocol P. We define $CC_t^{max}(f)$, the maximum communication complexity of f as $\min_P M(P)$ where P ranges over all deterministic protocol that outputs a correct answer. We have $CC_t^{max}(f) \geq \frac{1}{tR} CC_t^{tot}(f)$ where R is the number of rounds.

Let X be a subset of U^t where U is some finite universe and t is an integer. Define the **span** of X by $\mathsf{Span}(X) = \{y \in U^t | \forall\ i \in [t],\ \exists\ x \in X \text{ s. t. } y_i = x_i\}$. The notion k-fooling set introduced in [43] is defined as following.

Definition 4.1.1 (k-fooling set)**.** *Let $f : U^t \to \{0,1\}$ where U is some finite universe. Let $S \subseteq U^t$. For some integer k, we say S is a k-fooling set for f iff $f(x) = 0$ for each $x \in S$ and for each subset S' of S with cardinality k, the span of S' contains a member y such that $f(y) = 1$.*

We have the following.

Lemma 4.1.3 (Fact 4.1 from [43])**.** *Let S be a k-fooling set for f, we have $CC_t^{tot}(f) \geq \log(\frac{|S|}{k-1})$.*

4.2 Edit Distance

We show a reduction from the Set Disjointness problem (DIS) to computing ED between two strings in the asymmetric streaming model. For this, we define the following two party communication problem between Alice and Bob.

Given an alphabet Σ and three integers n_1, n_2, n_3. Suppose Alice has a string $x_1 \in \Sigma^{n_1}$ and Bob has a string $x_2 \in \Sigma^{n_1}$. There is another fixed reference string $y \in \Sigma^{n_3}$ that is known to both Alice and Bob. Alice and Bob now tries to compute $\mathsf{ED}((x_1 \circ x_2), y)$.We call this problem $\mathsf{ED}_{cc}(y)$. We prove the following theorem.

Lemma 4.2.1. *Suppose each input string to DIS has length n and let $\Sigma = [6n] \cup \{a\}$. Fix $y = (1, 2, \cdots, 6n)$. Then $R^{1/3}(\mathsf{ED}_{cc}(y)) \geq R^{1/3}(\mathsf{DIS})$.*

To prove this theorem, we first construct the strings x_1, x_2 based on the inputs $\alpha, \beta \in \{0,1\}^n$ to DIS. From α, Alice constructs the string $\alpha' \in \{0,1\}^{2n}$ such that $\forall j \in [n], \alpha'_{2j-1} = \alpha_j$ and $\alpha'_{2j} = 1 - \alpha_j$. Similarly, from β, Bob constructs the string $\beta' \in \{0,1\}^{2n}$ such that $\forall j \in [n], \beta'_{2j-1} = 1 - \beta_j$ and $\beta'_{2j} = \beta_j$. Now Alice lets x_1 be a modification from the string $(1, 2, 4, 5, \cdots, 3i-2, 3i-1, \cdots, 6n-2, 6n-1)$ such that $\forall j \in [2n]$, if $\alpha'_j = 0$ then the symbol $3j-1$ (at index $2j$) is replaced by a. Similarly, Bob lets x_2 be a modification from the string $(2, 3, 5, 6, \cdots, 3i-1, 3i, \cdots, 6n-1, 6n)$

such that $\forall j \in [2n]$, if $\beta'_j = 0$ then the symbol $3j-1$ (at index $2j-1$) is replaced by a.

We have the following Lemma.

Lemma 4.2.2. *If* $\mathsf{DIS}(\alpha, \beta) = 1$ *then* $\mathsf{ED}((x_1 \circ x_2), y) \geq 7n - 2$.

To prove the lemma we observe that in a series of edit operations that transforms (x_1, x_2) to y, there exists an index $r \in [6n]$ s.t. x_1 is transformed into $[1:r]$ and x_2 is transformed into $[r+1:n]$. We analyze the edit distance in each part. We first have the following claim:

Claim 4.2.1. *For any two strings u and v, there is a sequence of optimal edit operations (insertion/deletion/substitution) that transforms u to v, where all deletions happen first, followed by all insertions, and all substitutions happen at the end of the operations.*

Proof. Note that a substitution does not change the indices of the symbols. Thus, in any sequence of such edit operations, consider the last substitution which happens at index l. If there are no insertion/deletion after it, then we are good. Otherwise, consider what happens if we switch this substitution and the insertion/deletions before this substitution and after the second to last substitution. The only symbol that may be affected is the symbol where index l is changed into. Thus, depending on this position, we may or may not need a substitution, which results in a sequence of edit operations where the number of operations is at most the original number. In this way, we can change all substitutions to the end.

Further notice that we can assume without loss of generality that any deletion only deletes the original symbols in u, because otherwise we are deleting an inserted symbol, and these two operations cancel each other. Therefore, in a sequence of optimal edit operations, all the deletions can happen before any insertion. $\square$

For any i, let $\Gamma_1(i)$ denote the number of a symbols up to index $2i$ in x_1. Note that $\Gamma_1(i)$ is equal to the number of 0's in $\alpha'[1:i]$. We have the following lemma.

Lemma 4.2.3. *For any $p \in [n]$, let $r = 3p - q$ where $0 \leq q \leq 2$, then $\mathsf{ED}(x_1, [1:r]) = 4n - p - q + \Gamma_1(p)$ if $q = 0, 1$ and $\mathsf{ED}(x_1, [1:r]) = 4n - p + \Gamma_1(p-1)$ if $q = 2$.*

Proof. By Claim 4.2.1 we can first consider deletions and insertions, and then compute the Hamming distance after these operations (for substitutions).

We consider the three different cases of q. Let the number of insertions be d_i and the number of deletions be d_d. Note that $d_i - d_d = r - 4n$. We define the number of agreements between two strings to be the number of positions where the two corresponding symbols are equal.

The case of $q = 0$ and $q = 1$. Here again we have two cases.

Case (a): $d_d \geq 4n - 2p$. In this case, notice that the LCS after the operations between x_1 and y is at most the original $\mathsf{LCS}(x_1, y) = 2p - \Gamma_1(p)$. With d_i insertions, the number of agreements can be at most $\mathsf{LCS}(x_1, y) + d_i = 2p - \Gamma_1(p) + d_i$, thus the Hamming distance at the end is at least $r - 2p + \Gamma_1(p) - d_i$. Therefore, in this case the number of edit operations is at least $d_i + d_d + r - 2p + \Gamma_1(p) - d_i \geq 4n - p - q + \Gamma_1(p)$, and the equality is achieved when $d_d = 4n - 2p$.

Case (b): $d_d < 4n - 2p$. In this case, notice that all original symbols in x_1 larger than $3d_i$ (or beyond index $2d_i$ before the insertions) are guaranteed to be out of agreement. Thus the only possible original symbols in x_1 that are in agreement with y after the operations are the symbols with original index at most $2d_i$. Note that the LCS between $x_1[1:2d_i]$ and y is $2d_i - \Gamma_1(d_i)$. Thus with d_i insertions the number of agreements is at most $3d_i - \Gamma_1(d_i)$, and the Hamming distance at the end is at least $r - 3d_i + \Gamma_1(d_i)$.

Therefore the number of edit operations is at least $d_i + d_d + r - 3d_i + \Gamma_1(d_i) = r - d_i + (d_d - d_i) + \Gamma_1(d_i) = 4n - d_i + \Gamma_1(d_i)$. Now notice that $d_i = d_d + r - 4n < p$ and the quantity $d_i - \Gamma_1(d_i)$ is non-decreasing as d_i increases. Thus the number of edit operations is at least $4n - p + \Gamma_1(p) \geq 4n - p - q + \Gamma_1(p)$.

The other case of q is similar, as follows.

The case of $q = 2$. Here again we have two cases.

Case (a): $d_d \geq 4n - 2p + 1$. In this case, notice that the LCS after the operations between x_1 and y is at most the original $\mathsf{LCS}(x_1, y) = 2(p-1) - \Gamma_1(p-1) + 1 = 2p - 1 - \Gamma_1(p-1)$. With d_i insertions, the number of agreements can be at most $\mathsf{LCS}(x_1, y) + d_i = 2p - 1 - \Gamma_1(p-1) + d_i$, thus the Hamming distance at the end is at least $r - 2p + 1 + \Gamma_1(p-1) - d_i$. Therefore, in this case the number of edit operations is at least $d_i + d_d + r - 2p + 1 + \Gamma_1(p-1) - d_i \geq 4n - p + \Gamma_1(p-1)$, and the equality is achieved when $d_d = 4n - 2p + 1$.

Case (b): $d_d \leq 4n - 2p$. In this case, notice that all original symbols in x_1 larger than $3d_i$ (or beyond index $2d_i$ before the insertions) are guaranteed to be out of agreement. Thus the only possible original symbols in x_1 that are in agreement with y after the operations are the symbols with original index at most $2d_i$. Note that the LCS between $x[1 : 2d_i]$ and y is $2d_i - \Gamma_1(d_i)$. Thus with d_i insertions the number of agreements is at most $3d_i - \Gamma_1(d_i)$, and the Hamming distance at the end is at least $r - 3d_i + \Gamma_1(d_i)$.

Therefore the number of edit operations is at least $d_i + d_d + r - 3d_i + \Gamma_1(d_i) = r - d_i + (d_d - d_i) + \Gamma_1(d_i) = 4n - d_i + \Gamma_1(d_i)$. Now notice that $d_i = d_d + r - 4n < p - 1$ and the quantity $d_i - \Gamma_1(d_i)$ is non-decreasing as d_i increases. Thus the number of edit operations is at least $4n - (p-1) + \Gamma_1(p-1) > 4n - p + \Gamma_1(p-1)$.

□

We can now prove a similar lemma for x_2. For any i, let $\Gamma_2(i)$ denote the number of a symbols from index $2i+1$ to $4n$ in x_2. Note that $\Gamma_2(i)$ is equal to the number of 0's in $\beta'[i+1:2n]$.

Lemma 4.2.4. *Let $r = 3p+q$ where $0 \le q \le 2$, then $\mathsf{ED}(x_2, [r+1:6n]) = 2n+p-q+\Gamma_2(p)$ if $q = 0, 1$ and $\mathsf{ED}(x_2, [r+1:6n]) = 2n+p+\Gamma_2(p+1)$ if $q = 2$.*

Proof. We can reduce to Lemma 4.2.3. To do this, use $6n+1$ to minus every symbol in x_2 and in $[r+1:6n]$, while keeping all the a symbols unchanged. Now, reading both strings from right to left, x_2 becomes the string $\overline{x_2} = 1, 2, \cdots, 3i-2, 3i-1, \cdots, 6n-2, 6n-1$ with some symbols of the form $3j-1$ replaced by a's. Similarly $[r+1:6n]$ becomes $[1:6n-r]$ where $6n-r = 3(2n-p)-q$.

If we regard $\overline{x_2}$ as x_1 as in Lemma 4.2.3 and define $\Gamma_1(i)$ as in that lemma, we can see that $\Gamma_1(i) = \Gamma_2(2n-i)$.

Now the lemma basically follows from Lemma 4.2.3. In the case of $q = 0, 1$, we have

$$\mathsf{ED}(x_2, [r+1:6n]) = \mathsf{ED}(\overline{x'}, [1:6n-r]) = 4n-(2n-p)-q+\Gamma_1(2n-p) = 2n+p-q+\Gamma_2(p).$$

In the case of $q = 2$, we have

$$\mathsf{ED}(x_2, [r+1:6n]) = \mathsf{ED}(\overline{x'}, [1:6n-r]) = 4n-(2n-p)+\Gamma_1(2n-p-1) = 2n+p+\Gamma_2(p+1).$$

□

We can now prove Lemma 4.2.2.

Proof of Lemma 4.2.2. We show that for any $r \in [6n]$, $\mathsf{ED}(x_1, [1:r]) + \mathsf{ED}(x_2, [r+1:6n]) \ge 7n-2$. First we have the following claim.

Claim 4.2.2. *If* $\mathsf{DIS}(\alpha, \beta) = 1$*, then for any* $i \in [2n]$*, we have* $\Gamma_1(i) + \Gamma_2(i) \geq n$*.*

To see this, note that when i is even, we have $\Gamma_1(i) = i/2$ and $\Gamma_1(i) = n - i/2$ so $\Gamma_1(i) + \Gamma_2(i) = n$. Now consider the case of i being odd and let $i = 2j - 1$ for some $j \in [2n]$. We know $\Gamma_1(i-1) = (i-1)/2 = j - 1$ and $\Gamma_2(i+1) = n - (i+1)/2 = n - j$, so we only need to look at $x_1[2i-1 : 2i]$ and $x_2[2i+1 : 2i+2]$ and count the number of symbols a's in them. If the number of a's is at least 1, then we are done.

The only possible situation where the number of a's is 0 is that $\alpha'_i = \beta'_{i+1} = 1$ which means $\alpha_j = \beta_j = 1$ and this contradicts the fact that $\mathsf{DIS}(\alpha, \beta) = 1$.

We now have the following cases.

Case (a): $r = 3p$. In this case, by Lemma 4.2.3 and Lemma 4.2.4 we have $\mathsf{ED}(x_1, [1 : r]) = 4n - p + \Gamma_1(p)$ and $\mathsf{ED}(x_2, [r+1 : 6n]) = 2n + p + \Gamma_2(p)$. Thus we have $\mathsf{ED}(x_1, [1 : r]) + \mathsf{ED}(x_2, [r+1 : 6n]) = 6n + n = 7n$.

Case (b): $r = 3p - 1 = 3(p-1) + 2$. In this case, by Lemma 4.2.3 and Lemma 4.2.4 we have $\mathsf{ED}(x_1, [1 : r]) = 4n - p - 1 + \Gamma_1(p)$ and $\mathsf{ED}(x_2, [r+1 : 6n]) = 2n + (p-1) + \Gamma_2(p)$, thus we have $\mathsf{ED}(x_1, [1 : r]) + \mathsf{ED}(x_2, [r+1 : 6n]) = 6n - 2 + n = 7n - 2$.

Case (c): $r = 3p - 2 = 3(p-1) + 1$. In this case, by Lemma 4.2.3 and Lemma 4.2.4 we have $\mathsf{ED}(x_1, [1 : r]) = 4n - p + \Gamma_1(p-1)$ and $\mathsf{ED}(x_2, [r+1 : 6n]) = 2n + (p-1) - 1 + \Gamma_2(p-1)$, thus we have $\mathsf{ED}(x_1, [1 : r]) + \mathsf{ED}(x_2, [r+1 : 6n]) = 6n - 2 + n = 7n - 2$.

□

We now prove Lemma 4.2.1.

Proof of Lemma 4.2.1. We begin by upper bounding $\mathsf{ED}((x_1 \circ x_2), y)$ when $\mathsf{DIS}(\alpha, \beta) = 0$.

Claim 4.2.3. *If* $\mathsf{DIS}(\alpha, \beta) = 0$ *then* $\mathsf{ED}((x_1 \circ x_2), y) \leq 7n - 3$*.*

To see this, note that if $\mathsf{DIS}(\alpha, \beta) = 0$ then there exists a $j \in [n]$ such that $\alpha_j = \beta_j = 1$. Thus $\alpha'_{2j-1} = 1$, $\beta'_{2j-1} = 0$ and $\alpha'_{2j} = 0$, $\beta'_{2j} = 1$. Note that the number of 0's in $\alpha'[1 : 2j - 1]$ is $j - 1$ and thus $\Gamma_1(2j - 1) = j - 1$. Similarly the number of 0's in $\beta'[2j : 2n]$ is $n - j$ and thus $\Gamma_2(2j - 1) = n - j$. To transform (x_1, x_2) to y, we choose $r = 6j - 2$, transform x_1 to $y[1 : r]$, and transform x_2 to $y[r + 1 : 6n]$.

By Lemma 4.2.3 and Lemma 4.2.4 we have $\mathsf{ED}(x_1, [1 : r]) = 4n - 2j + \Gamma_1(2j - 1)$ and $\mathsf{ED}(x_2, [r + 1 : 6n]) = 2n + (2j - 1) - 1 + \Gamma_2(2j - 1)$. .Thus we have $\mathsf{ED}(x_1, [1 : r]) + \mathsf{ED}(x_2, [r + 1 : 6n]) = 6n - 2 + \Gamma_1(2j - 1) + \Gamma_2(2j - 1) = 6n - 2 + n - 1 = 7n - 3$. Therefore $\mathsf{ED}((x_1, x_2), y) \leq 7n - 3$.

Therefore, in the case of $\mathsf{DIS}(\alpha, \beta) = 1$, we have $\mathsf{ED}((x_1 \circ x_2), y) \geq 7n - 2$ while in the case of $\mathsf{DIS}(\alpha, \beta) = 0$, we have $\mathsf{ED}((x_1 \circ x_2), y) \leq 7n - 3$. Thus any protocol that solves $\mathsf{ED}_{cc}(y)$ can also solve DIS, hence the theorem follows. $\square$

In the proof of Lemma 4.2.1, the two strings $x = (x_1 \circ x_2)$ and y have different lengths, however we can extend it to the case where the two strings have the same length and prove the following theorem.

Lemma 4.2.5. *Suppose each input string to* DIS *has length* n *and let* $\Sigma = [16n] \cup \{a\}$. *Fix* $\tilde{y} = (1, 2, \cdots, 16n, a^{2n})$*, let* $\tilde{x}_1 \in \Sigma^{4n}$ *and* $\tilde{x}_2 \in \Sigma^{14n}$. *Define* $\mathsf{ED}_{cc}(\tilde{y})$ *as the two party communication problem of computing* $\mathsf{ED}((\tilde{x}_1 \circ \tilde{x}_2), \tilde{y})$. *Then* $R^{1/3}(\mathsf{ED}_{cc}(\tilde{y})) \geq R^{1/3}(\mathsf{DIS})$.

Proof. We extend the construction of Lemma 4.2.1 as follows. From input (α, β) to DIS, first construct (x_1, x_2) as before. Then, let $z = (6n + 1, 6n + 2, \cdots, 16n)$, $\tilde{x}_1 = x_1$ and $\tilde{x}_2 = x_2 \circ z$. Note that we also have $\tilde{y} = y \circ z \circ a^{2n}$, and $|\tilde{x}_1 \circ \tilde{x}_2| = 18n = |\tilde{y}|$.

We finish the proof by establishing the following two lemmas.

Lemma 4.2.6. *If* $\mathsf{DIS}(\alpha, \beta) = 1$ *then* $\mathsf{ED}((\tilde{x}_1 \circ \tilde{x}_2), \tilde{y}) \geq 9n - 2$.

Proof. First we can see that $\mathsf{ED}((\tilde{x}_1 \circ \tilde{x}_2), \tilde{y}) < 10n$ since we can first use at most

$8n-1$ edit operations to change (x_1, x_2) into y (note that the first symbols are the same), and then add $2n$ symbols of a at the end.

Now we have the following claim:

Claim 4.2.4. *In an optimal sequence of edit operations that transforms $(\tilde{x}_1 \circ \tilde{x}_2)$ to $\tilde{y}$, at the end some symbol in z must be kept and thus matched to $\tilde{y}$ at the same position.*

To see this, assume for the sake of contradiction that none of the symbols in z is kept, then this already incurs at least $10n$ edit operations, contradicting the fact that $\mathsf{ED}((\tilde{x}, \tilde{x}'), \tilde{y}) < 10n$.

We now have a second claim:

Claim 4.2.5. *In an optimal sequence of edit operations that transforms $(\tilde{x}_1 \circ \tilde{x}_2)$ to $\tilde{y}$, at the end all symbols in z must be kept and thus matched to $\tilde{y}$ at the same positions.*

To see this, we use Claim 4.2.4 and first argue that some symbol of z is kept and matched to $\tilde{y}$. Assume this is the symbol r. Then we can grow this symbol both to the left and to the right and argue that all the other symbols of z must be kept. For example, consider the symbol $r+1$ if $r < 16n$. There is a symbol $r+1$ that is matched to $\tilde{y}$ in the end. If this symbol is not the original $r+1$, then the original one must be removed either by deletion or substitution, since there cannot be two symbols of $r+1$ in the end. Thus instead, we can keep the original $r+1$ symbol and reduce the number of edit operations.

More precisely, if this $r+1$ symbol is an insertion, then we can keep the original$r+1$ symbol and get rid of this insertion and the deletion of the original $r+1$, which saves 2 operations. If this $r+1$ symbol is a substitution, then we can keep the original$r+1$ symbol, delete the symbol being substituted, and get rid of the deletion of the original $r+1$, which saves 1 operation. The case of $r-1$ is completely symmetric. Continue doing this, we see that all symbols of z must be kept and thus matched to $\tilde{y}$.

Now, we can see the optimal sequence of edit operations must transform (x_1, x_2) into y, and transform the empty string into a^{2n}. Thus by Lemma 4.2.2 we have

$$\mathsf{ED}((\tilde{x}_1 \circ \tilde{x}_2), \tilde{y}) = \mathsf{ED}((x_1, x_2), y) + 2n \geq 9n - 2.$$

□

We now have the next lemma.

Lemma 4.2.7. *If* $\mathsf{DIS}(\alpha, \beta) = 0$ *then* $\mathsf{ED}((\tilde{x}_1 \circ \tilde{x}_2), \tilde{y}) \leq 9n - 3$.

Proof. Again, to transform $(\tilde{x}_1 \circ \tilde{x}_2)$ to $\tilde{y}$, we can first transform (x_1, x_2) into y, and insert a^{2n} at the end. If $\mathsf{DIS}(\alpha, \beta) = 0$ then by Claim 4.2.3 $\mathsf{ED}((x_1 \circ x_2), y) \leq 7n - 3$. Therefore we have

$$\mathsf{ED}((\tilde{x}_1 \circ \tilde{x}_2), \tilde{y}) \leq \mathsf{ED}((x_1, x_2), y) + 2n \leq 9n - 3.$$

Thus any protocol that solves $\mathsf{ED}_{cc}(\tilde{y})$ can also solve DIS, hence the theorem follows. □

□

From Lemma 4.2.5 we immediately have the following theorem.

Lemma 4.2.8. *Any R-pass randomized algorithm in the asymmetric streaming model that computes $\mathsf{ED}(x, y)$ exactly between two strings x, y of length n with success probability at least $2/3$ must use space at least $\Omega(n/R)$.*

We can generalize the theorem to the case of deciding if $\mathsf{ED}(x, y)$ is a given number k. First we prove the following lemmas.

Lemma 4.2.9. *Let Σ be an alphabet. For any $n, \ell \in \mathbb{N}$ let $x, y \in \Sigma^n$ and $z \in \Sigma^\ell$ be three strings. Then $\mathsf{ED}(x \circ z, y \circ z) = \mathsf{ED}(x, y)$.*

Proof. First it is clear that $\mathsf{ED}(x \circ z, y \circ z) \leq \mathsf{ED}(x, y)$, since we can just transform x to y. Next we show that $\mathsf{ED}(x \circ z, y \circ z) \geq \mathsf{ED}(x, y)$.

To see this, suppose a series of edit operations transforms x to $y' = y[1 : n - r]$ or $y' = y[1 : n] \circ z[1 : r]$ for some $r \geq 0$ and transforms z to the other part of $y \circ z$ (called z'). Then by triangle inequality we have $\mathsf{ED}(x, y') \geq \mathsf{ED}(x, y) - r$. Also note that $\mathsf{ED}(z, z') \geq ||z| - |z'|| = r$. Thus the number of edit operations is at least $\mathsf{ED}(x, y') + \mathsf{ED}(z, z') \geq \mathsf{ED}(x, y)$. □

Lemma 4.2.10. *Let Σ be an alphabet. For any $n, \ell \in \mathbb{N}$ let $x, y \in \Sigma^n$ and $u, v \in \Sigma^\ell$ be four strings. If there is no common symbol between any of the three pairs of strings (u, v), (u, y) and (v, x), then $\mathsf{ED}(x \circ u, y \circ v) = \mathsf{ED}(x, y) + \ell$.*

Proof. First it is clear that $\mathsf{ED}(x \circ u, y \circ v) \leq \mathsf{ED}(x, y) + \ell$, since we can just transform x to y and then replace u by v. Next we show that $\mathsf{ED}(x \circ u, y \circ v) \geq \mathsf{ED}(x, y) + \ell$.

To see this, suppose a series of edit operations transforms x to $y' = y[1 : n - r]$ for some $r \geq 0$ and transforms u to the other part of $v' = y[n - r + 1 : n] \circ v$. Then by triangle inequality we have $\mathsf{ED}(x, y') \geq \mathsf{ED}(x, y) - r$. Since there is no common symbol between (u, v) and (u, y) , we have $\mathsf{ED}(u, v') \geq r + \ell$. Thus the number of edit operations is at least $\mathsf{ED}(x, y') + \mathsf{ED}(u, v') \geq \mathsf{ED}(x, y) + \ell$. The case of transforming x to $y' = y[1 : n] \circ v[1 : r]$ for some $r \geq 0$ is completely symmetric since equivalently it is transforming y to $x' = [1 : n - r']$ for some $r' \geq 0$. □

We have the following two theorems.

Lemma 4.2.11. *There is a constant $c > 1$ such that for any $k, n \in \mathbb{N}$ with $n \geq ck$, and alphabet Σ with $|\Sigma| \geq ck$, any R-pass randomized algorithm in the asymmetric streaming model that decides if $\mathsf{ED}(x, y) \geq k$ between two strings $x, y \in \Sigma^n$ with success probability at least $2/3$ must use space at least $\Omega(k/R)$.*

Proof. Lemma 4.2.5 and Lemma 4.2.8 can be viewed as deciding if $\mathsf{ED}(x, y) \geq 9n - 2$

for two strings of length $18n$ over an alphabet with size $16n+1$. Thus we can first use the constructions there to reduce DIS to the problem of deciding if $\mathsf{ED}(x, y) \geq k$ with a fixed string y of length $O(k)$. The number of symbols used is $O(k)$ as well. Now to increase the length of the strings to n, we pad a sequence of the symbol 1 at the end of both x and y until the length reaches n. By Lemma 4.2.9 the edit distance stays the same and thus the problem is still deciding if $\mathsf{ED}(x, y) \geq k$. By Lemma 4.2.5 the communication complexity is $\Omega(k)$ and thus the theorem follows. □

Lemma 4.2.12. *There is a constant $c > 1$ such that for any $k, n \in \mathbb{N}$ and alphabet Σ with $n \geq ck \geq |\Sigma|$, any R-pass randomized algorithm in the asymmetric streaming model that decides if $\mathsf{ED}(x, y) \geq k$ between two strings $x, y \in \Sigma^n$ with success probability at least $2/3$ must use space at least $\Omega(|\Sigma|/R)$.*

Proof. Lemma 4.2.5 and Lemma 4.2.8 can be viewed as deciding if $\mathsf{ED}(x, y) \geq 9n-2$ for two strings of length n over an alphabet with size $18n+1$. Thus we can first use the constructions there to reduce DIS to the problem of deciding if $\mathsf{ED}(x, y) \geq k' = \Omega(|\Sigma|)$ with a fixed string y of length $\Theta(|\Sigma|)$, and the number of symbols used is $|\Sigma| - 2$. Now we take the 2 unused symbols and pad a sequence of these two symbols with length $k - k'$ at the end of x and y. By Lemma 4.2.10 the edit distance increases by $k - k'$ and thus the problem becomes deciding if $\mathsf{ED}(x, y) \geq k$. Next, to increase the length of the strings to n, we pad a sequence of the symbol 1 at the end of both strings until the length reaches n. By Lemma 4.2.9 the edit distance stays the same and thus the problem is still deciding if $\mathsf{ED}(x, y) \geq k$. By Lemma 4.2.5 the communication complexity is $\Omega(|\Sigma|)$ and thus the theorem follows. □

Combining the previous two lemmas we have the following theorem, which is a restatement of Theorem 4.1.

Theorem 4.1. *There is a constant $c > 1$ such that for any $k, n \in \mathbb{N}$ with $n \geq ck$, given an alphabet Σ, any R-pass randomized algorithm in the asymmetric streaming* *model that decides if* $\mathsf{ED}(x,y) \geq k$ *for two strings* $x, y \in \Sigma^n$ *with success probability* $\geq 2/3$ *must use space* $\Omega(\mathsf{min}(k, |\Sigma|)/R)$*.*

For $0 < \varepsilon < 1$, by taking $k = 1/\varepsilon$ we also get the following corollary:

Corollary 4.2.1. *Given an alphabet* Σ*, for any* $0 < \varepsilon < 1$*, any R-pass randomized algorithm in the asymmetric streaming model that achieves a* $1+\varepsilon$ *approximation of* $\mathsf{ED}(x,y)$ *between two strings* $x, y \in \Sigma^n$ *with success probability at least* $2/3$ *must use space at least* $\Omega(\mathsf{min}(1/\varepsilon, |\Sigma|)/R)$*.*

4.3 Longest Common Subsequence

In this section, we study the space lower bounds for asymmetric streaming LCS.

4.3.1 Lower Bounds for Exact Computation

We first show a lower bound for instances where the input strings are over binary alphabet.

Binary Alphabet, Deterministic Algorithm For binary strings, we can show a $\Omega(n)$ lower bound. In this proof, we assume n can be diveded by 60 and let $l = \frac{n}{30} - 1$. We assume the alphabet is $\Sigma = \{a, b\}$. Consider strings x of the form

$$x = b^{10}a^{s_1}b^{10}a^{s_2}b^{10}\cdots b^{10}a^{s_l}b^{10}. \tag{4.1}$$

That is, x contains l blocks of consecutive a symbols. Between each block of a symbols, we insert 10 b's and we also add 10 b's to the front, and the end of x. $s_1, \ldots, s_l$ are l integers such that

$$\sum_{i=1}^{l} s_i = \frac{n}{6} + 5, \tag{4.2}$$
$$1 \le s_i \le 9,\ \forall i \in [l]. \tag{4.3}$$

Thus, the length of x is $\sum_{i=1}^{l} \frac{n}{6} + 5 + 10(l+1) = \frac{n}{2} + 5$ and it contains exactly $\frac{n}{3}$ b's.

Let S be the set of all $x \in \{a,b\}^{\frac{n}{2}+5}$ of form 4.1 that satisfying equations 4.2, 4.3. For each string $x \in S$, we can define a string $f(x) \in \{a,b\}^{\frac{n}{2}-5}$ as following. Assume $x = b^{10}a^{s_1}b^{10}a^{s_2}b^{10}\cdots b^{10}a^{s_l}b^{10}$, we set $f(x) = a^{s_1}b^{10}a^{s_2}b^{10}\cdots b^{10}a^{s_l}b^{10}$. That is, $f(x)$ simply removed the first 10 b's of x. We denote $\bar{S} = \{f(x)|x \in S\}$.

Claim 4.3.1. $|S| = |\bar{S}| = 2^{\Omega(n)}$.

Proof. Notice that for $x^1, x^2 \in S$, if $x^1 \neq x^2$, then $f(x^1) \neq f(x^2)$. We have $|S| = |\bar{S}|$.

The size of S equals to the number of choices of l integers $s_1, s_2, \dots, s_l$ that satisfies 4.2 and 4.3. For an lower bound of $|S|$, we can pick $\frac{n}{60}$ of the integers to be 9, and set the remaining to be 1 or 2. Thus the number of such choices is at least $\binom{l}{\frac{n}{60}} = \binom{\frac{n}{30}-1}{\frac{n}{60}} = 2^{\Omega(n)}$.

□

We first show the following lemma.

Lemma 4.3.1. *Let $y = a^{n/3}b^{n/3}a^{n/3}$. For every $x \in S$,*

$$\mathit{LCS}(x \circ f(x), y) = \frac{n}{2} + 5.$$

For any two distinct $x^1, x^2 \in S$,

$$\max\{\mathit{LCS}(x^1 \circ f(x^2), y), \mathit{LCS}(x^2 \circ f(x^1), y)\} > \frac{n}{2} + 5.$$

Proof of Lemma 4.3.1. We first show $\mathsf{LCS}(x \circ f(x), y) = \frac{n}{2} + 5$. Notice that $x \circ f(x)$ is of the form

$$b^{10}a^{s_1}b^{10} \cdots b^{10}a^{s_l}b^{10}a^{s_1}b^{10} \cdots b^{10}a^{s_l}b^{10}.$$

It cantains $2l + 1$ block of b's, each consists 10 consecutive b's. These blocks of b's are seperated by some a's. Also, $x \circ f(x)$ has $2\sum_{i=1}^{l} s_i = \frac{n}{3} + 10$ a's and $\frac{2n}{3} - 10$ b's. Let p_i be the first position of the i-th block of b's.

Let us consider a matching between $x \circ f(x)$ and y.

If the matching does not match any b's in y to $x \circ f(x)$, the size of such a matching is at most $\frac{n}{3} + 10$ since it is the number of a's in $x \circ f(x)$.

Now we assume some 1's in y are matched to $x \circ f(x)$. Without loss of generality, we can assume the first b symbol in y is matched. This is because all b's in y are consecutive and if the first b in y (i.e. $y_{\frac{n}{3}+1}$) is not matched, we can find another matching of the same size that matches $y_{\frac{n}{3}+1}$. For the same reason, we can assume the first b in y is matched to position p_i for some $i \in [2l + 1]$. Assume n_b is the number of b's matched. Again, without loss of generality, we can assume the first n_b b's starting from position p_i in $x \circ f(x)$ are matched since all b's in y are consecutive and there are no matched a's between two matched b's. We have two cases.

Case 1: $y_{\frac{n}{3}+1}$ is matched to p_i for some $1 \le i \le l + 1$. Let n_b be the number of matched b's. We know $n_b \le \frac{n}{3}$ since there are $\frac{n}{3}$ b's in y.

If $n_b = \frac{n}{3}$, we match first $\frac{n}{3}$ b's in $x \circ f(x)$ starting from position p_i. Consider the number of a's that are still free to match in $x \circ f(x)$. The number of a's before p_i is $\sum_{j=1}^{i-1} s_i$. Since $i \le l + 1$, $\sum_{j=1}^{i-1} s_i$ is at most $\frac{n}{6} + 5$, we can match all of them to first third of y. Also, we need $l + 1$ blocks of b's to match all b's in y. The number of a's after last matched b in $x \circ f(x)$ is $\sum_{j=i}^{l} s_j$ (which is zero when $i = l + 1$). Again, we can match all these a's since $\sum_{j=i}^{l} s_j \le \frac{n}{3}$. In total, we can match $\sum_{j=1}^{l} s_j = \frac{n}{6} + 5$ a's. This gives us a matching of size $\frac{n}{3} + \frac{n}{6} + 5 = \frac{n}{2} + 5$.

We argue that the best strategy is to always match $\frac{n}{3}$ b's. To see this, if we removed 10 matched b's, this will let us match s_j additional 0's for some $j \in [l]$. By our construction of x, s_j is strictly smaller than 10 for all $j \in [l]$. If we keep doing this, the size of matching will decrease. Thus, the largest matching we can find in this case is $\frac{n}{2} + 5$.

Case 2: $y_{\frac{n}{3}+1}$ is matched to p_i for some $l + 1 < i \leq 2l + 1$. By the same argument in Case 1, the best strategy is to match as many b's as possible. The number of b's in $x \circ f(x)$ starting from position p_i is $(2l + 1 - i + 1)t = (2l - i + 2)t$. The number of a's that are free to match is $\sum_{j=1}^{l} s_j + \sum_{j'=1}^{i-l+1} s_{j'} = \frac{n}{6} + 5 + \sum_{j=1}^{i-l-1} s_j$. Since $s_j < t$ for all $j \in [l]$, the number of a's can be matched is strictly smaller than $\frac{n}{6} + 5 + (i - l + 1)t$. The largest matching we can find in this case is smaller than $\frac{n}{6} + 5 + (l + 1)t = \frac{n}{2} + 5$.

This proves the size of the largest matching we can find is exactly $\frac{n}{2} + 5$. We have $\mathsf{LCS}(x \circ f(x), y) = \frac{n}{2} + 5$.

For the second statement in the lemma, say x^1 and x^2 are two distinct strings in S. For convenience, we assume $x^1 = b^{10}a^{s_1^1}b^{10}a^{s_2^1}b^{10} \cdots b^{10}a^{s_l^1}b^{10}$, and $x^2 = b^{10}a^{s_1^2}b^{10}a^{s_2^2}b^{10} \cdots b^{10}a^{s_l^2}b^{10}$. Let i be the smallest integer such that $s_i^1 \neq s_2^i$. We have $i \leq l - 1$ since $\sum_{j=1}^{l} s_j^1 = \sum_{j=1}^{l} s_j^2$. Without loss of generality, we assume $s_i^1 > s_i^2$. We show that $\mathsf{LCS}(x^1 \circ f(x^2), y) > \frac{n}{2} + 5$. Notice that $x^1 \circ f(x^2)$ is of the form

$$b^{10}a^{s_1^1}b^{10} \cdots b^{10}a^{s_l^1}b^{10}a^{s_1^2}b^{10} \cdots b^{10}a^{s_l^2}b^{10}.$$

By the same notation, let p_j be the first position of the j-th block of b's in $x^1 \circ f(x^2)$

Consider the match that matches the first b in y to position p_{i+1}. The number of a's before p_{i+1} is $\sum_{j=1}^{i} s_j^1$. We matches all $\frac{n}{3}$ b's in y. The number of a's after the last matched b in $x^1 \circ f(x^2)$ is $\sum_{j=i+1}^{l} s_j^2$. This gives us a match of size $\frac{n}{3} + \sum_{j=1}^{i} s_j^1 + \sum_{j=i+1}^{l} s_j^2$. By our choice of i, we have $s_j^1 = s_j^2$ for $j \in [i-1]$. The size of the matching equals to

$\frac{n}{3} + \sum_{j=1}^{l} s_j^2 + s_i^1 - s_i^2 = \frac{n}{2} + 5 + s_i^1 - s_i^2$ which is larger than $\frac{n}{2} + 5$. Thus, the length of LCS between $x^1 \circ f(x^2)$ and y is larger than $\frac{n}{2} + 5$. This finishes our proof.

□

Lemma 4.3.2. *In the asymmetric streaming model, any deterministic protocol that computes $\mathsf{LCS}(x, y)$ for any $x, y \in \{0, 1\}^n$, in R passes of x needs $\Omega(n/R)$ space.*

Proof. Consider a two party game where player 1 holds a string $x^1 \in S$ and player 2 holds a string $x^2 \in S$. The goal is to verify whether $x^1 = x^2$. It is known that the total communication complexity of testing the equality of two elements from set S is $\Omega(\log |S|)$, see [125] for example. We can reduce this to computing the length of LCS. To see this, we first compute $\mathsf{LCS}(x^1 \circ f(x^2), y)$ and $\mathsf{LCS}(x^2 \circ f(x^1), y)$ with $y = a^{n/3}b^{n/3}a^{n/3}$. By lemma 4.3.1, if both $\mathsf{LCS}(x^1 \circ f(x^2), y) = \mathsf{LCS}(x^2 \circ f(x), y) = \frac{n}{2} + 5$, we know $x^1 = x^2$, otherwise, $x^1 \neq x^2$. Here, y is known to both parties.

The above reduction shows the total communication complexity of this game is $\Omega(n)$ since $|S| = 2^{\Omega(n)}$. If we only allow R rounds of communication, the size of the longest message sent by the players is $\Omega(n/R)$. Thus, in the asymmetric model, any protocol that computes $\mathsf{LCS}(x, y)$ in R passes of x needs $\Omega(n/R)$ space.

□

The fooling-set construction presented above is an important building block for our lower bounds in Section 4.3.2.

$\Omega(n)$ size alphabet, randomized algorithm

Lemma 4.3.3. *Assume $|\Sigma| = \Omega(n)$. In the asymmetric streaming model, any randomized protocol that computes $\mathsf{LCS}(x, y)$ correctly with probability at least $2/3$ for any $x, y \in \Sigma^n$, in R of passes of x needs $\Omega(n/R)$ space. The lower bound also holds when x is a permutation of y.*

Proof of Lemma 4.3.3. The proof is by giving a reduction from set-disjointness.

In the following, we assume alphabet set $\Sigma = [n]$ which is the set of integers from 1 to n. We let the online string x be a permutation of n and the offline string $y = 12 \cdots n$ be the concatenation from 1 to n. Then computing $\mathsf{LCS}(x, y)$ is equivalent to compute $LIS(x)$ since any common subsequence of x and y must be increasing.

We now describe our approach. For convenience, let $n' = \frac{n}{2}$. Without loss of generality, we assume n' can be divided by 4. Consider a string $z \in \{0,1\}^{n'}$ with the following property.

$$\forall \ i \in [\frac{n'}{2}], \qquad z_{2i} = 1 - z_{2i-1}. \tag{4.4}$$

For each $i \in [n']$, we consider subsets $\sigma^i \subset [n]$ for $i \in [n']$ as defined below

$$\sigma^i = \begin{cases} \{4i-1, 4i\}, & \text{if } i \text{ is odd and } i \le \frac{n'}{2} \\ \{4(i-\frac{n'}{2})-3, 4(i-\frac{n'}{2})-2\} & \text{if } i \text{ is odd and } i > \frac{n'}{2} \\ \{4i-3, 4i-2\} & \text{if } i \text{ is even and } i \le \frac{n'}{2} \\ \{4(i-\frac{n'}{2})-1, 4(i-\frac{n'}{2})\} & \text{if } i \text{ is even and } i > \frac{n'}{2} \end{cases}$$

Notice that $\sigma^i \cap \sigma^j = \emptyset$ if $i \neq j$ and $\cup_{i=1}^{n'} \sigma^i = [n]$. For an odd $i \in [n'/2]$, $\min \sigma^i > \max \sigma^{i+n'/2}$. Oppositely, for an even $i \in [n'/2]$, $\max \sigma^i < \min \sigma^{i+n'/2}$. Also notice that for distinct $i, j \in [n'/2]$ with $i < j$, $\max \sigma^i < \min \sigma^j$ and $\max \sigma^{i+n'/2} < \min \sigma^{j+n'/2}$.

We abuse the notation a bit and let $\sigma^i(z_i)$ be a string such that if $z_i = 1$, the string consists of elements in set σ^i arranged in an increasing order, and if $z_i = 0$, the string is arranged in decreasing order.

Let x be the concatenation of $\sigma^i(z_i)$ for $i \in [n']$ such that $x = \sigma^1(z_1) \circ \sigma^2(z_2) \circ \cdots \circ \sigma^{n'}(z_{n'})$. By the definition of σ^i's, we know x is a permutation of $[n]$.

For convenience, let z^1 and z^2 be two subsequences of z such that

$$z^1 = z_2 \circ z_4 \circ \cdots z_{\frac{n'}{2}}$$

$$z^2 = z_{\frac{n'}{2}+2} \circ z_{\frac{n'}{2}+4} \circ \cdots \circ z_{n'}$$

If $\mathsf{DIS}(z^1, z^2) = 0$. Then there exist some $i \in [n'/4]$ such that $z_{2i} = z_{n'/2+2i} = 1$. Notice that in z, we have $\forall j \in [\frac{n'}{2}],\ z_{2j} = 1 - z_{2j-1}$. Thus, $\mathsf{LIS}(\sigma^{2j-1}(z_{2j-1}) \circ \sigma^{2j}(z_{2j})) = 3$ since only one of $\sigma^{2j-1}(z_{2j-1})$ and $\sigma^{2j}(z_{2j})$ is increasing and the other is decreasing. For any $j \in [n'/4]$, we have

$$\mathsf{LIS}\left(\sigma^1(z_1) \circ \sigma^2(z_2) \circ \cdots \circ \sigma^{2j}(z_{2j})\right) = 3j, \tag{4.5}$$

$$\mathsf{LIS}\left(\sigma^{2j+n'/2+1}(z_{2j+n'/2+1}) \circ \sigma^{2j+n'/2+2}(z_{2j+n'/2+2}) \circ \cdots \circ \sigma^{n'}(z_{n'})\right) = 3\frac{n'-2j}{2}. \tag{4.6}$$

Since $z_{2i+n'/2} = 1$, $\mathsf{LIS}(\sigma^{2i+n'/2}(z_{2i+n'/2})) = 2$. Since $\max \sigma^{2i} < \min \sigma^{2i+n'/2} < \max \sigma^{2i+n'/2} < \min \sigma^{2i+n'/2+1}$, combining with equations 4.5 and 4.6, we know $\mathsf{LIS}(x) \geq 3\frac{n'}{2} + 2$.

If $\mathsf{DIS}(z^1, z^2) = 1$, we prove that $\mathsf{LIS}(x) = 3\frac{n'}{2} + 1$. We only need to consider in an longest increasing subsequence, when do we first pick some elements from the second half of x. Say the first element picked in the second half of x is in $\sigma^{2i+n'/2-1}$ or $\sigma^{2i+n'/2}$ for some $i \in [n'/2]$. We have

$$\mathsf{LIS}\left(\sigma^{2i-1}(z_{2i-1}) \circ \sigma^{2i}(z_{2i}) \circ \sigma^{2i+n'/2-1}(z_{2i+n'/2-1}) \circ \sigma^{2i+n/2}(z_{2i+n'/2})\right) = 4.$$

This is because z_{2i} and $z_{2i+n'/2}$ can not both be 1, $z_{2i} = 1 - z_{2i-1}$, and $z_{2i+n'/2} = 1 - z_{2i+n'/2-1}$. The length of LIS of the substring of x before $\sigma^{2i-1}(z_{2i-1})$ is $3(i-1)$ and the length of LIS after $\sigma^{2i+n/2}(z_{2i+n'/2})$ is $3(n'/2 - i)$. Thus, we have $\mathsf{LIS}(x) = 3\frac{n'}{2} + 1$.

This gives a reduction from computing $\mathsf{DIS}(z^1, z^2)$ to computing $\mathsf{LIS}(x) = \mathsf{LCS}(x, y)$. Now assume player 1 holds the first half of x and player 2 holds the second half. Both players have access to y. Since $|z^1| = |z^2| = n'/4 = \frac{n}{8}$. Any randomized protocol that computes $\mathsf{LCS}(x, y)$ with success probability at least 2/3 has an total communication complexity $\Omega(n)$. Thus, any randomized asymmetric streaming algorithm with R passes of x needs $\Omega(n/R)$ space. □

Theorem 4.2 is a generalization of Lemma 4.3.3. Below is a formal proof.

Proof of Theorem 4.2. Without loss of generality, assume $\Sigma = [r]$ so $|\Sigma| = r$. Since we assume $k < n$, we have $\mathsf{min}(k, r) < n$.

Let $d = \mathsf{min}(k, r)$. we let the offline string y be the concatenation of two parts y^1 and y^2 where y^1 is the concatenation of symbols in $[d-2] \subseteq \Sigma$ in ascending order. y^2 is the symbol $d-1$ repeated $n-d+2$ times. Thus, $y \in \Sigma^n$. x also consists of two parts x^1 and x^2 such that x^1 is over alphabet $[d-2]$ with length $d-2$ and x^2 is the symbol d repeated $n-d+2$ times. Since the symbol $d-1$ does not appear in x and the symbol d does not appear in y. Thus, $\mathsf{LCS}(x, y) = \mathsf{LCS}(x^1, y^1)$. By Lemma 4.3.3, any randomized algorithm that computes $\mathsf{LCS}(x^1, y^1)$ with probability at least 2/3 using R passes of x^1 requires $\Omega(d/R)$ space.

□

4.3.2 Lower Bounds for Approximation

In this section, we prove Theorem 4.3.

Proof of Theorem 4.3. In the following, we assume the size of alphabet set Σ is $3r$ such that

$$\Sigma = \{a_1, b_1, c_1, a_2, b_2, c_2, \dots a_r, b_r, c_r\}.$$

We let $n_\varepsilon = \Theta(1/\varepsilon)$ such that n_ε can be divided by 60.

For any distinct $a, b \in \Sigma$, we can build a set $S_{a,b}$ in the same way as we did in section 4.3.1 except that we replace n wtih n_ε (we used notation S instead $S_{a,b}$). Thus, $S_{a,b} \subseteq \{a, b\}^{n_\varepsilon/2+5}$. Similarly, we can define function f and $\overline{S_{a,b}} = \{f(\sigma) | \sigma \in S_{a,b}\} \subset \{a, b\}^{n_\varepsilon/2-5}$. Let $y^{a,b} = a^{n_\varepsilon/3} b^{n_\varepsilon/3} a^{n_\varepsilon/3}$. By Claim 4.3.1 and Lemma 4.3.1, we know $|S_{a,b}| = |\overline{S_{a,b}}| = 2^{\Omega(n_\varepsilon)}$. For any $\sigma \in S_{a,b}$, we have $\mathsf{LCS}(\sigma \circ f(\sigma), y^{a,b}) = n_\varepsilon/2 + 5$. For any two distinct $\sigma^1, \sigma^2 \in S_{a,b}$, we know at least one of $\mathsf{LCS}(\sigma^1 \circ f(\sigma^2), y^{a,b})$ and $\mathsf{LCS}(\sigma^2 \circ f(\sigma^1), y^{a,b})$ is at larger than $n_\varepsilon/2 + 5$.

Consider $w = (w_1, w_2, \dots, w_{2r})$ such that $w_{2i-1} \in S_{a,b}$ and $w_{2i} \in \overline{S_{a,b}}$ for $i \in [r]$.

Thus, w can be viewed as an element in

$$U = \underbrace{\left(S_{a,b} \times \overline{S_{a,b}}\right) \times \cdots \times \left(S_{a,b} \times \overline{S_{a,b}}\right)}_{r \text{ times}}.$$

For alphabet $\{a_i, b_i\}$, we can similarly define S_{a_i,b_i}, $\overline{S_{a_i,b_i}}$ and y^{a_i,b_i}. We let U_i be similarly defined as U but over alphabet $\{a_i, b_i\}$.

Let $\beta = 1/3$. We can define function $h : (S_{a,b} \times \overline{S_{a,b}})^r \to \{0,1\}$ such that

$$h(w) = \begin{cases} 0, & \text{if } \forall i \in [r],\ \mathsf{LCS}(w_{2i-1} \circ w_{2i}, y^{a_i,b_i}) = n_\varepsilon/2 + 5 \\ 1, & \text{if for at least } \beta r \text{ indices } i \in [r],\ \mathsf{LCS}(w_{2i-1} \circ w_{2i}, y^{a,b}) > n_\varepsilon/2 + 5 \\ \text{undefined}, & \text{otherwise.} \end{cases} \tag{4.7}$$

Consider an error-correcting code $T_{a,b} \subseteq S_{a,b}^r$ over alphabet $S_{a,b}$ with constant rate α and constant distance β. We can pick $\alpha = 1/2$ and $\beta = 1/3$, for example. Then the size of the code $T_{a,b}$ is $|T_{a,b}| = |S_{a,b}|^{\alpha r} = 2^{\Omega(n_\varepsilon r)}$. For any code word $\chi = \chi_1\chi_2 \cdots \chi_r \in T_{a,b}$ where $\chi_i \in S_{a,b}$, we can write

$$\nu(\chi) = (\chi_1, f(\chi_1), \chi_2, f(\chi_2), \ldots, \chi_r, f(\chi_r)).$$

Let $W = \{\nu(\chi) | \chi \in T_{a,b}\} \in (S_{a,b} \times \overline{S_{a,b}})^r$. Then

$$|W| = |T_{a,b}| = 2^{\Omega(n_\varepsilon r)} \tag{4.8}$$

We consider a $2r$-player one-way game where the goal is to compute the function h on input $w = (w_1, w_2, \ldots, w_{2r})$. In this game, player i holds w_i for $i \in [2r]$ and can only send message to player $i+1$ (Here, $2r + 1 = 1$). We now show the following claim.

Claim 4.3.2. *W is a fooling set for the function h.*

Proof. Consider any two different codewords $\chi, \chi' \in T_{a,b}$. Denote $w = \nu(\chi)$ and $w' = \nu(\chi')$. By our defintion of w, we know $w_{2i} = f(w_{2i-1})$ and $w_{2i-1} \in S_{a_i,b_i}$ for

$i \in [r]$. We have $\mathsf{LCS}(w_{2i-1} \circ w_{2i}, y^{a_i,b_i}) = n_\varepsilon/2 + 5$. Thus, $h(w) = h(w') = 0$.

The span of $\{w, w'\}$ is the set $\{(v_1, v_2, \ldots, v_{2r}) | v_i \in \{w_i, w'_i\}$ for $i \in [2r]\}$. We need to show that there exists some v in the span of $\{w, w'\}$ such that $h(v) = 1$. Since χ, χ' are two different codewords of $T_{a,b}$. We know there are at least βr indices i such that $w_{2i-1} \neq w'_{2i-1}$. Let $I \subseteq [r]$ be the set of indices that $\chi_i \neq \chi'_i$. Then, for $i \in I$, we have

$$\max\Big(\mathsf{LCS}(w_{2i-1} \circ w'_{2i}, y^{a_i,b_i}), \mathsf{LCS}(w'_{2i-1} \circ w_{2i}, y^{a_i,b_i}))\Big) \geq n_\varepsilon + 6.$$

We can build a v as following. For $i \in I$, if $\mathsf{LCS}(w_{2i-1} \circ w'_{2i}, y^{a,b} \geq n_\varepsilon + 6$. We then set $v_{2i-1} = w_{2i-1}$ and $v_{2i} = w'_{2i}$. Otherwise, we set $v_{2i-1} = w'_{2i-1}$ and $v_{2i} = w_{2i}$. For $i \in [r] \setminus I$, we set $v_{2i-1} = w_{2i-1}$ and $v_{2i} = w_{2i}$. Thus, for at least βr indices $i \in [r]$, we have $\mathsf{LCS}(v_{2i-1} \circ v_{2i}, y^{a_i,b_i}) \geq n_\varepsilon/2 + 6$. We must have $h(v) = 1$. □

Consider a matrix B of size $r \times 2r$ of the following form

$$\begin{pmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,2r} \\ B_{2,1} & B_{2,2} & \cdots & B_{2,2r} \\ \vdots & \vdots & \ddots & \vdots \\ B_{r,1} & B_{r,2} & \cdots & B_{r,2r} \end{pmatrix}$$

where $B_{i,2j-1} \in S_{a_i,b_i}$ and $B_{i,2j} \in \overline{S_{a_i,b_i}}$ for $j \in [r]$ (the elements of matrix B are strings over alphabet Σ). Thus, the i-th row of B is an element in U_i. We define the following function g.

$$g(B) = h_1(R_1(B)) \vee h_2(R_2(B)) \vee \cdots \vee h_r(R_r(B)) \tag{4.9}$$

where $R_i(B) \in U$ is the i-th row of matrix B and h_i is the same as h except the inputs are over alphabet $\{a_i, b_i\}$ instead of $\{a, b\}$ for $i \in [r]$. Also, we define W_i in exactly the same way as W except elements in W_i are over alphabet $\{a_i, b_i\}$ instead of $\{a, b\}$.

Consider a $2t$ player game where each player holds one column of B. The goal is to compute $g(B)$. We first show the following Claim.

Claim 4.3.3. *The set of all $r \times 2r$ matrix B such that $R_i(B) \in W_i \ \forall \ i \in [r]$ is a fooling set for g.*

Proof of Claim 4.3.3. For any two matrix $B_1 \neq B_2$ such that $R_i(B_1), R_i(B_2) \in W_i \ \forall \ i \in [r]$. We know $g(B_1) = g(B_2) = 0$. There is some row i such that $R_i(B_1) \neq R_i(B_2)$. We know there is some elements v in the span of $R_i(B_1)$ and $R_i(B_2)$, such that $h_i(v) = 1$ by Claim 4.3.2. Thus, there is some element B' in the span of B_1 and B_2 such that $g(B') = 1$. Here, by B' in the span of B_1 and B_2, we mean the i-th column is either $C_i(B_1)$ or $C_i(B_2)$. □

Since $|W| = 2^{\Omega(n_\varepsilon r)}$. By the above claim, we have a fooling set for g size $|W|^r = 2^{\Omega(n_\varepsilon r^2)}$. Thus, $CC^{tot}_{2r}(g) \geq \log(2^{\Omega(n_\varepsilon r^2)}) = \Omega(n_\varepsilon r^2)$. Since $CC^{\max}_{2r} \geq CC^{tot})_{2r}(g) = \Omega(n_\varepsilon r)$.

We now show how to reduce computing $g(B)$ to approximating the length of LCS in the asymmetric streaming model.

We consider a matrix $\tilde{B}$ of size $r \times 3r$ such that

$$\tilde{B} = \begin{pmatrix} B_{1,1} & B_{1,2} & c_1^{n_\varepsilon} & B_{1,3} & B_{1,4} & c_2^{n_\varepsilon} & \cdots & B_{1,2r-1} & B_{1,2r} & c_r^{n_\varepsilon} \\ B_{2,1} & B_{2,2} & c_1^{n_\varepsilon} & B_{2,3} & B_{2,4} & c_2^{n_\varepsilon} & \cdots & B_{2,2r-1} & B_{2,2r} & c_r^{n_\varepsilon} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ B_{r,1} & B_{r,2} & c_1^{n_\varepsilon} & B_{r,3} & B_{r,4} & c_2^{n_\varepsilon} & \cdots & B_{r,2r-1} & B_{r,2r} & c_r^{n_\varepsilon} \end{pmatrix}_{(r \times 3r)} \tag{4.10}$$

In other words, $\tilde{B}$ is obtained by inserting a column of c symbols to B at every third position. For $j \in [3r]$, let $C_j(\tilde{B}) = \tilde{B}_{1,j} \circ \tilde{B}_{2,j} \circ \cdots \circ \tilde{B}_{r,j}$. That is, $C_j(\tilde{B})$ is the concatenation of elements in the j-th column of $\tilde{B}$. We can set x to be the concatenation of the columns of $\tilde{B}$. Thus, since $B_{i,2j-1} \circ B_{i,2j} \in \Sigma^{n_\varepsilon}$ for any $i, j \in [r]$, we have

$$x = C_1(\tilde{B}) \circ C_2(\tilde{B}) \circ \cdots \circ C_{3r}(\tilde{B}) \in \Sigma^{2r^2 n_\varepsilon}.$$

For $i \in r$, we have defined $y^{a_i,b_i} = a_i^{n_\varepsilon/3} b_i^{n_\varepsilon/3} a_i^{n_\varepsilon/3} \in \Sigma^{n_\varepsilon}$. We let $y^{i,1}$ and $y^{i,2}$ be two non-empty strings such that $y^{a_i,b_i} = y^{i,1} \circ y^{i,2}$. We consider another matrix $\bar{B}$ of size $r \times 3r$ such that

$$\bar{B} = \begin{pmatrix} y^{1,1} & y^{1,2} & c_1^{n_\varepsilon} & y^{1,1} & y^{1,2} & c_2^{n_\varepsilon} & \cdots & y^{1,1} & y^{1,2} & c_r^{n_\varepsilon} \\ y^{2,1} & y^{2,2} & c_1^{n_\varepsilon} & y^{2,1} & y^{2,2} & c_2^{n_\varepsilon} & \cdots & y^{2,1} & y^{2,2} & c_r^{n_\varepsilon} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ y^{r,1} & y^{r,2} & c_1^{n_\varepsilon} & y^{r,1} & y^{r,2} & c_2^{n_\varepsilon} & \cdots & y^{r,1} & y^{r,2} & c_r^{n_\varepsilon} \end{pmatrix}_{(r \times 3r)} \tag{4.11}$$

For $i \in [r]$, let $R_i(\bar{B})$ be the concatenation of elements in the i-th row of $\bar{B}$. We can set y to be the concatenation of rows of $\bar{B}$. Thus,

$$y = R_1(\bar{B}) \circ R_2(\bar{B}) \circ \cdots \circ C_r(\bar{B}) \in \Sigma^{2r^2 n_\varepsilon}.$$

We now show the following Claim.

Claim 4.3.4. *If $g(B) = 0$, $\mathsf{LCS}(x, y) \leq (\frac{5}{2}r - 1)n_\varepsilon + 5r$. If $g(B) = 1$, $\mathsf{LCS}(x, y) \geq (\frac{5}{2}r - 1)n_\varepsilon + 5r + \beta r - 1$.*

Proof. We first divide x into r blocks such that

$$x = x^1 \circ x^2 \circ \cdots x^r$$

where $x^i = C_{3i-2}(\tilde{B}) \circ C_{3i-1}(\tilde{B}) \circ C_{3i}(\tilde{B})$. We know $C_{3i-2}(\tilde{B}) = C_{2i-1}(B)$ is the $(2i-1)$-th column of B, $C_{3i-1}(\tilde{B}) = C_{2i}(B)$ is the $2i$-th column of B and $C_{3i}(\tilde{B}) = c_i^{rn_\varepsilon}$ is symbol c_i repeated rn_ε times.

If $g(B) = 0$, we show $\mathsf{LCS}(x, y) \leq (\frac{5}{2}r - 1)n_\varepsilon + 5r$. We consider a matching between x and y. For our analysis, we let t_i be the largest integer such that some symbols in x^i is matched to symbols in the t_i-th row of $\bar{B}$. Since y is the concatenation of rows of $\bar{B}$. If no symbols in x^i is matched, we let $t_i = t_{i-1}$ and we set $t_0 = 1$, we have

$$1 = t_0 \leq t_1 \leq t_2 \leq \cdots \leq t_r \leq r$$

We now show that, there is an optimal matching such that x^i is matched to at most $n_\varepsilon/2 + 5 + (t_i - t_{i-1} + 1)n_\varepsilon$ symbols in y. There are two cases:

Case (a): $t_i = t_{i-1}$. In this case, x^i can only be matched to t_i-th row of $\bar{B}$, $R_{t_i}(\bar{B})$. $R_{t_i}(\bar{B})$ consists of symbols a_{t_i}, b_{t_i} and $c_1, \ldots, c_r$ and is of the form

$$R_{t_i}(\bar{B}) = y^{a_{t_i}, b_{t_i}} \circ c_1^{n_\varepsilon} \circ y^{a_{t_i}, b_{t_i}} \circ c_2^{n_\varepsilon} \circ \cdots \circ y^{a_{t_i}, b_{t_i}} \circ c_r^{n_\varepsilon}.$$

We first show that we can assume a_{t_i} and b_{t_i} symbols in x^i are only matched to the $y^{a_{t_i}, b_{t_i}}$ block between the block of c_{i-1} and the block of c_i.

If in $R_{t_i}(\bar{B})$, there are some a_{t_i}, b_{t_i} symbols before the block of c_{i-1} matched to x^i. Say the number of such matches is at most n'. We know $n' \leq n_\varepsilon$ there are at most n_ε a_{t_i}, b_{t_i} symbols in x^i. In this case, notice that, there is no c_{i-1} symbol in $R_{t_i}(\bar{B})$ can be matched to x. We can build another matching between x and y by removing these n' matches and add n_ε matches between c_{i-1} symbols in x^{i-1} and $R_{t_i}(\bar{B})$. We can do this since there are rn_ε c_{i-1} symbols in x^{i-1} and before the t_i-th row, we can match at most $(t_i - 1)n_\varepsilon$ c_{i-1} symbols. So there are at least $(r - t_i + 1)n_\varepsilon$ unmatched c_{i-1} symbols in x^{i-1}. The size of the new matching is at least the size of the old matching.

Similarly, if there are some a_{t_i}, b_{t_i} symbols after the block of c_i in $R_{t_i}(y)$ matched to x^i. Then, no c_i symbol in x^i can be matched to $R_{t_i}(\bar{B})$. We can remove these matches and add n_ε matched c_i symbols. This gives us a matching with size at least the size fo the old matching.

Thus, we only need to consider the case where $B_{t_i,2i-1} \circ B_{t_i,2i}$ is matched to the part of $R_{t_i}(\bar{B})$ after the block of c_{i-1} symbols and before the block of c_i symbols, which is $y^{a_{t_i},b_{t_i}}$. Since $g(B) = 0$, we know $\mathsf{LCS}(B_{t_i,2i-1} \circ B_{t_i,2i}, y^{a_{t_i},b_{t_i}})$ is exactly $n_\varepsilon/2 + 5$. Also, we can match at most n_ε c_i symbols. Thus, x^i is matched to at most $n_\varepsilon/2 + 5 + n_\varepsilon$ symbols in y.

Case (b): $t_i > t_{i-1}$. We can assume in x^i, except c_i symbols, only symbols $a_{t_{i-1}}, b_{t_{i-1}}$ are matched to y. To see this, assume t' with $t_{i-1} < t' \le t_i$ is the largest integer that some symbol $a_{t'}$ or $b_{t'}$ in x^i are matched to y. By a, b symbols, we mean symbols $a_1, \dots, a_r$ and $b_1, \dots, b_r$. We now consider how many a, b symbols in x^i can be matched to y. We only need to consider the substring

$$B_{t_{i-1},2i-1} \circ B_{t_{i-1}+1,2i-1} \circ \cdots \circ B_{t',2i-1} \circ B_{t_{i-1},2i} \circ B_{t_{i-1}+1,2i} \circ \cdots \circ B_{t',2i}$$

Let $t_{i-1} \le k \le t'$ be the largest integer such that some symbol a_k or b_k from $B_{k,2i-1}$ is matched. Notice that for any $t \in [r]$, symbol a_t, b_t only appears in the t-th row of $\bar{B}$ and y is the concatenation of row of $\bar{B}$. For a_t, b_t with $t < k$, only those in $B_{t,2i-1}$ can be matched since block $B_{t,2i}$ is after $B_{k,2i-1}$ in x^i. For a_t, b_t, with $t > k$, only those in $B_{t,2i}$ can be matched by assumption on k.

Notice that for any t, we have $B_{t,2i-1}$ has length $n_\varepsilon/2 + 5$ and $B_{t,2i}$ has length $n_\varepsilon/2 - 5$. Thus, the number of matched a, b symbols is at most $(t' - t_{i-1})(n_\varepsilon/2 + 5) + n_\varepsilon$. We can build another matching by first remove the matches of a, b symbols in x^i. Then, we match another $(t' - t_{i-1})n_\varepsilon$ c_{i-1} symbols in x^{i-1} to the c_{i-1} symbols in $\bar{B}$ from $(t_{i-1} + 1)$-th row to t'-th row. These c_{i-1} symbols are not matched since we assume x^{i-1} is only matched to first t_{i-1} rows of $\bar{B}$ in the original matching. Further, we can match $B_{t',2i-1} \circ B_{t',2i}$ to the $y^{a_{t'},b_{t'}}$ block in $R_{t'}(\bar{B})$ between the block of c_{i-1} and the block of c_i. This gives us $(t' - t_{i-1})n_\varepsilon + n_\varepsilon/2 + 5$ additional matches. Since we $t' > t_{i-1}$, we know the number of added matches is at least the number of removed matches. In the new matching, $t_{i-1} = t'$. Thus, we can assume in x^i, except c_i symbols, only symbols $a_{t_{i-1}}, b_{t_{i-1}}$ are matched to y.

By the same argument in Case (a), we can assume a_{t_i} and b_{t_i} symbols in x^i are only matched to the $y^{a_{t_i}, b_{t_i}}$ block between the block of c_{i-1} and the block of c_i. Thus, we can match $n_\varepsilon/2 + 5$ a_{t_i} and b_{t_i} symbols. Also, we can match at most $(t_i - t_{i-1} + 1)n_\varepsilon$ c_i symbols since there are this many c_i symbols in $\bar{B}$ from t_{i-1}-th row to t_i-th row. Thus, x^i is matched to at most $n_\varepsilon/2 + 5 + (t_i - t_{i-1} + 1)n_\varepsilon$ symbols in y.

In total, the size of matching is at most

$$\sum_{i=1}^{r} \Big(n_\varepsilon/2 + 5 + (t_i - t_{i-1} + 1)n_\varepsilon\Big) \leq (2r-1)n_\varepsilon + (n_\varepsilon/2 + 5)r = (\frac{5}{2}r - 1)n_\varepsilon + 5r$$

Thus, if $g(B) = 0$, we know $\mathsf{LCS}(x, y) \leq (\frac{5}{2}r - 1)n_\varepsilon + 5r$

If $g(B) = 1$, that means there is some row i of B such that for at least βr positions $j \in [r]$, we have $\mathsf{LCS}(B_{i,2j-1} \circ B_{i,2j}, y^{a_i, b_i}) \geq n_\varepsilon/2 + 6$.

We now build a mathing between x and y with size at least $(\frac{5}{2}r - 1)n_\varepsilon + 5r + \beta r - 1$. We first match $B_{1,1} \circ B_{1,2}$, which is a subsequence of $C_1(\tilde{B}) \circ C_2(\tilde{B})$, to the first y^{a_1, b_1} block in $R_1(\bar{B})$, this gives us at least $n_\varepsilon/2 + 5$ matches. Then, we match all the c_1 symbols in the first i rows of $\bar{B}$ to $C_3\tilde{B}$. This gives us in_ε matches.

We consider the string

$$\tilde{x} = B_{i,3} \circ B_{i,4} \circ \circ c_2^{n_\varepsilon} \circ \cdots \circ B_{i,2r-1} \circ B_{i,2r} \circ c_r^{n_\varepsilon}.$$

It is a subsequence of $x^2 \circ \cdots x^r$. Also, for at least $\beta r - 1$ positions $j \in \{2, 3, \ldots, r\}$, we know $\mathsf{LCS}(B_{i,2j-1} \circ B_{i,2j}, y^{a_i, b_i}) \geq n_\varepsilon/2 + 6$. For the rest of the positions, we know $\mathsf{LCS}(B_{i,2j-1} \circ B_{i,2j}, y^{a_i, b_i}) = n_\varepsilon/2 + 5$. Thus, $\mathsf{LCS}(\tilde{x}, R_i(\bar{B})) \geq (r-1)\Big(n_\varepsilon + (n_\varepsilon/2 + 5)\Big) + \beta r - 1$.

After the i-th row of $\bar{B}$, we can match another $(r-i)n_\varepsilon$ c_r symbols to x^r. This gives us a matching of size at least $(\frac{5}{2}r - 1)n_\varepsilon + 5r + \beta r - 1$. Thus, if $g(B) = 1$,

$\mathsf{LCS}(x,y) \geq (\frac{5}{2}r-1)n_\varepsilon + 5r + \beta r - 1$.

□

Assume $n_\varepsilon = \lambda/\varepsilon$ where λ is some constant. Let $\varepsilon' = \frac{\beta}{10\lambda}\varepsilon = \Theta(\varepsilon)$. If we can give a $1+\varepsilon'$ approximation of $\mathsf{LCS}(x,y)$, we can distinguish $g(B)=0$ and $g(B)=1$.

Thus, we can reduce computing $g(B)$ in the $2r$ player setting to computing $\mathsf{LCS}(x,y)$. The string y (the offline string) is known to all players and it contains no information about B. For the i-th player, it holds i-th column of B. If i is odd, the player i knows the $3\frac{i-1}{2}$-th column of $\tilde{B}$. If i is even, the player i knows the $(\frac{3i}{2}-1)$-th row of $\tilde{B}$ which is the i-th row of B and $(\frac{3i}{2})$-th row of $\tilde{B}$ which consist of only $c_{\frac{3i}{2}}$.

□

4.4 Longest Increasing/Non-decreasing Subsequence

In this section, we proof our space lower bound for LIS and LNS.

Let $a \in \{0,1\}^t$, a can be seen as a binary string of length t. For each integer $l \geq 1$, we can define a function $h^{(l)}$ whose domain is a subset of $\{0,1\}^t$. Let $\alpha \in (1/2, 1)$ be some constant. We have following definition

$$h^{(l)}(a) = \begin{cases} 1, & \text{if there are at least } l \text{ zeros between any two nonzero positions in } a. \\ 0, & \text{if } a \text{ contains at least } \alpha t \text{ nonzeros.} \end{cases} \tag{4.12}$$

We leave $h^{(l)}$ undefined otherwise. Let $B \in \{0,1\}^{s\times t}$ be a matrix and denote the i-th row of B by $R_i(B)$. We can define $g^{(l)}$ as the direct sum of s copies of $h^{(l)}$. Let

$$g^{(l)}(B) = h^{(l)}(R_1(B)) \vee h^{(l)}(R_2(B)) \vee \cdots \vee h^{(l)}(R_s(B)). \tag{4.13}$$

That is, $g^{(l)}(B) = 1$ if there is some $i \in [s]$ such that $h^{(l)}(R_i(B)) = 1$ and $g^{(l)}(B) = 0$ if for all $i \in [s]$, $h^{(l)}(R_i(B)) = 0$.

In the following, we consider computing $h^{(l)}$ and $g^{(l)}$ in the t-party one-way communication model. When computing $h^{(l)}(a)$, player P_i holds the i-th element of $a \in \{0,1\}^t$ for $i \in [t]$. When computing $g^{(l)}(B)$, player P_i holds the i-th column of matrix B for $i \in [t]$. In the following, we use $CC_t^{tot}(h^{(l)})$ to denote the total communication complexity of $h^{(l)}$ and respectively use $CC_t^{tot}(g^{(l)})$ to denote the total communication complexity of $g^{(l)}$. We also consider multiple rounds of communication and we denote the number of rounds by R.

Lemma 4.4.1. *For any constant $l \geq 1$, there exists a constant k (depending on l), such that there is a k-fooling set for function $h^{(l)}$ of size c^t for some constant $c > 1$.*

We note that Lemma 4.2 of [43] proved a same result for the case $l = 1$.

Proof of Lemma 4.4.1. We consider sampling randomly from $\{0,1\}^t$ as follows. For $i \in [t]$, we independently pick $a_i = 1$ with probability p and $a_i = 0$ with probability $1-p$. We set $p = \frac{1}{k}$ for some large constant k. For $i \in [t-l]$, we let A_i to be event that there are no two 1's in the substring $a[i : i+l]$. Let $\Pr(A_i)$ be the probability that event A_i happens. By a union bound, we have

$$\Pr(A_i) \leq \binom{l}{2} p^2, \ \forall\, i \in [t-l] \tag{4.14}$$

Let $\mathcal{A} = \{A_i,\ i \in [t-l]\}$. Notice that since we are sampling each position of a independently, the event A_i is dependent to at most $2l$ other events in $\mathcal{A}$. We set $v = 2l\binom{l}{2}p^2$. For large enough k, we have

$$\forall\, i \in [t-l],\ \Pr(A_i) \leq \binom{l}{2} p^2 \leq v(1-v)^{2l} \tag{4.15}$$

Here, the second inequality follows from the fact that l is a constant and $\binom{l}{2}p^2 = v/(2l)$, so we can pick k to be large enough, say $k \geq \sqrt{\frac{l^3}{1-log(2l)/(2l)}}$ (or $p = 1/k$ to be small enough) to guarantee $(1-v)^{2l} \geq 1/2l$.

Thus, we can use Lovás Local Lemma here. By Lemma 4.1.1, we have

$$\Pr\left(\overline{A_1} \wedge \overline{A_2} \wedge \cdots \wedge \overline{A_{t-l}}\right) \geq (1-v)^t \geq (1-l^3p^2)^t. \tag{4.16}$$

Notice that "there are at least l 0's between any two 1's in a" is equivalent to none of events A_i happens. We say a sampled string a is good if none of A_i happens. Thus, for any good string a, we have $h^{(l)}(a) = 0$. The probability that a sampled string a is good is at least $(1-l^3p^2)^t$. For convenience, we let $q = 1 - l^3p^2$.

Assume we independently sample M strings in this way, the expected number good string is q^tM. Let $a^1, a^2, \dots, a^k$ be k independent random samples. We consider a string b in the span of these k strings, such that, for $i \in [t]$, let $b_i = 1$ if there is some $j \in [k]$ such that $a_i^j = 1$. b is in the span of these k strings. We now consider the probability that b has at least αt 1's, i.e. $h^{(l)}(b) = 1$. Notice that $a_i^j = 1$ with probability p, thus $Pr(b_i = 1) = 1 - (1-p)^t$. Let $\gamma = 1 - (1-p)^t$. The expected number of 1's in b is γt. Let $\varepsilon = \gamma - \alpha$ and δ be the probability that b has less than αt 1's. Using Chernoff bound, we have,

$$\Pr(b \text{ has less than } \alpha t \text{ 1's}) \leq e^{\frac{-\varepsilon^2\gamma}{2}t}. \tag{4.17}$$

Let $\delta = e^{\frac{-\varepsilon^2\gamma}{2}t}$ and $M = \frac{e}{k}(\frac{q^t}{2\delta})^{1/k}$. We consider the probability that these M sample is not a k-fooling set for $h^{(l)}$. Since for any k samples, it is not a k-fooling set with probability at most δ. Let E denote the event that these M samples form a k-fooling set. Using a union bound, the probability that E does not happen is

$$\Pr\left(\bar{E}\right) \leq \binom{M}{k}\delta \leq (\frac{eM}{k})^k\delta \leq \frac{1}{2}q^t.$$

Let Z be a random variable equals to the number of good string among the M samples. As we have shown, the expectation of Z, $\mathbb{E}(Z) = q^tM$. Also notice that $\mathbb{E}(Z) = \mathbb{E}(Z|E)\Pr(E) + \mathbb{E}(Z|\overline{E})\Pr(\overline{E})$. Thus, with a positive probability, there are $\frac{1}{2}q^tM$ good samples and they form a k-fooling set.

$$\frac{1}{2}q^tM = (\frac{1}{2})^{1+1/k}\frac{e}{k}(q^{1+1/k}(\frac{1}{\delta})^{1/k})^t. \tag{4.18}$$

Notice that

$$q^{1+1/k}(\frac{1}{\delta})^{1/k} = (1 - \frac{l^3}{k^2})^{1+1/k} e^{\frac{\varepsilon^2 \gamma}{2k}}. \tag{4.19}$$

Since we assume l is a constant, it is larger than 1 when k is a constant large enough (depends on l). This finishes the proof.

$\square$

The following lemma is essentially the same as Lemma 4.3 in [43].

Lemma 4.4.2. *Let $F \subseteq \{0,1\}^t$ be a k-fooling set for $h^{(l)}$. Then the set of all matrix $B \in \{0,1\}^{s\times t}$ such that $R_i(B) \in F$ is a k^s-fooling set for $g^{(l)}$.*

Lemma 4.4.3. $CC_t^{max}(g^{(l)}) = \Omega(s/R)$.

Proof. By Lemma 4.4.1 and Lemma 4.4.2, there is a k^s-fooling set for function $g^{(l)}$ of size c^{ts} for some large enough constant k and some constant $c > 1$. By Lemma 4.1.3, in the t-party one-way communication model, $CC_t^{tot}(g^{(l)}) = \Omega(\log \frac{c^{ts}}{k^s - 1}) = \Omega(ts)$. Thus, we have $CC_t^{max}(g^{(l)}) \geq \frac{1}{tR} CC_t^{tot}(g^{(l)}) = \Omega(s/R)$. $\square$

4.4.1 Lower bound for streaming LIS over small alphabet

We now present our space lower bound for approximating LIS in the streaming model.

Lemma 4.4.4. *For $x \in \Sigma^n$ with $|\Sigma| = O(\sqrt{n})$ and any constant $\varepsilon > 0$, any deterministic algorithm that makes R passes of x and outputs a $(1+\varepsilon)$-approximation of $\mathsf{LIS}(x)$ requires $\Omega(|\Sigma|/R)$ space.*

Proof of Lemma 4.4.4. We assume the alphabet set $\Sigma = \{0, 1, \dots, 2r\}$ which has size $|\Sigma| = 2r + 1$. Let c be a large constant and assume r can be divided by c for similicity. We set $s = \frac{r}{c}$ and $t = r$. Consider a matrix B of size $s \times t$. We denote the element on i-th row and j-th column by $B_{i,j}$. ALso, we require that $B_{i,j}$ is either $(i-1)\frac{r}{c} + j$ or 0. For each row of B, say $R_i(B)$, either there are at least l 0's between any two nonzeros or it has more than αr nonzeros. We let $\tilde{B} \in \{0,1\}^{s\times r}$ be a binary matrix such that $\tilde{B}_{i,j} = 1$ if $B_{i,j} \neq 0$ and $\tilde{B}_{i,j} = 0$ if $B_{i,j} = 0$ for $(i,j) \in [s] \times [r]$.

Without loss of generality, we can view $R_i(B)$ for $i \in [s]$, or $C_i(B)$ for $i \in [r]$ as a string. More specifically, $R_i(B) = B_{i,1}B_{i,2}\dots B_{i,r}$ for $i \in [s]$, and $C_i(B) = B_{1,i}B_{2,i}\dots B_{s,i}$ for $i \in [r]$.

We let $\sigma(B) = C_1(B) \circ C_2(B) \circ \dots \circ C_r(B)$. Thus, $\sigma(B)$ is a string of length sr. For convenience, we denote $\sigma = \sigma(B)$. Here, we require the length of $\sigma = r^2/c \leq n$. If $|\sigma| < n$, we can pad σ with 0 symbols to make it has length n. This will not affect the length of the longest increasing subsequence of σ.

We first show that if there is some row of B that contains more than αt nonzeros, then $\mathsf{LIS}(\sigma) \geq \alpha r$. Say $R_i(B)$ contains more than αt nonzeros. By our definition of B, $R_i(B)$ is strictly increasing when restricted to the nonzero positions. Thus, $\mathsf{LIS}(R_i(B)) \geq \alpha r$. Also notice that $R_i(B)$ is a subsequence of σ. This is because $C_j(B)$ contains element $B_{i,j}$ for $j \in [r]$ and σ is the concatenation of $C_j(B)$'s for j from 1 to r. Thus, $\mathsf{LIS}(\sigma) \geq \alpha r$.

Otherwise, for any row $R_i(B)$, there are at least l zeros between any two nonzero positions. We show that $\mathsf{LIS}(\sigma(B)) \leq (\frac{1}{r} + \frac{1}{c})r$. Assume $\mathsf{LIS}(\sigma) = m$ and let $\sigma' = B_{p_1,q_1}B_{p_2,q_2}\dots B_{p_m,q_m}$ be a longest increasing subsquence of σ.

We can think of σ' as a path on the matrix. By go down k steps from (i,j), we mean walk from (i,j) to $(i+k,j)$. By go right k steps form (i,j), we mean walk from (i,j) to $(i,j+k)$. Then σ' corresponds to the path $P(\sigma') = (p_1,q_1) \to (p_2,q_2) \to \dots \to (p_m,q_m)$. Notice that for each step $(p_i,q_i) \to (p_{i+1},q_{i+1})$, we know $1 \leq B_{p_i,q_i} < B_{p_{i+1},q_{i+1}}$. Thus, by our construction of matrix B, the step can only be one of the three types:

Type 1: $p_i < p_{i+1}$ and $q_{i+1} \geq q_i$. If $p_i < p_{i+1}$, then for any $q_{i+1} \geq q_i$, we have $B_{p_i,q_i} < B_{p_{i+1},q_{i+1}}$ if B_{p_i,q_i} and $B_{p_{i+1},q_{i+1}}$ are both non zero,

Type 2: $p_i = p_{i+1}$ and $q_{i+1} - q_i \geq l + 1$.This correspons to the case where B_{p_i,q_i}

and $B_{p_{i+1},q_{i+1}}$ are picked from the same row of B. However, since we assume in each row of B, the number of 0's between any two nonzeros is at least l. Since B_{p_i,q_i} and $B_{p_{i+1},q_{i+1}}$ are both nonzeros and $B_{p_i,q_i} < B_{p_{i+1},q_{i+1}}$, we must have $q_{i+1} - q_i \geq l + 1$.

Type 3: $p_i > p_{i+1}$ and $q_{i+1} - q_i \geq (p_i - p_{i+1})\frac{r}{c} + 1$. When $p_i > p_{i+1}$, $B_{p_i,q_i} - B_{p_{i+1},q_i} > (p_i - p_{i+1})\frac{r}{c}$. Since we require $B_{p_i,q_i} < B_{p_{i+1},q_{i+1}}$, we must have $q_{i+1} - q_i \geq (p_i - p_{i+1})\frac{r}{c} + 1$.

For $i \in [3]$, let a_i be the number of step of Type i in the path $P(\sigma')$. Then, $a_1 + a_2 + a_3 = m - 1$. We say $(p_i, q_i) \to (p_{i+1}, q_{i+1})$ is step i for $i \in [m-1]$. And let

$$U_i = \{j \in [m-1] | \text{step } j \text{ is of Type } i\} \text{ for } i \in [3].$$

$$a_1' = \sum_{i \in U_1} (p_{i+1} - p_i)$$
$$a_3' = \sum_{i \in U_3} (p_i - p_{i+1}).$$

Or equivalently, a_1' is the distance we go downward with steps of Type 1 and a_2' is the distance we go upward with steps of Type 3. Since only steps of Type 1 and Type 3 can go up and down, we know

$$a_1' - a_3' = p_m - p_1 \leq \frac{r}{c} \tag{4.20}$$

For the number of distance we go right, for each step of Type 2, we go at least $l+1$ positions right. For the step of Type 3, we go right at least $\sum_{i \in T_3}((p_i - p_{i+1})\frac{r}{c} + 1) = \frac{r}{c}a_3' + |T_3| = \frac{r}{c}a_3' + a_3$ steps. Since the total distance we go right is $q_m - q_1 \leq r$. Thus, we have

$$a_2 l + a_3' \frac{r}{c} + a_3 \leq q_m - q_1 \leq r. \tag{4.21}$$

We assume l and c are both constants and $\frac{r}{cl} \geq 1$. Notice that $a_1 \leq a_1'$ and $a_3 \leq a_3'$, combining 4.20 and 4.21, we have

$$\mathsf{LIS}(\sigma(B)) = a_1 + a_2 + a_3 + 1 \leq \frac{r}{c} + \frac{r}{l}.$$

This show that if $g^{(l)}(\tilde{B}) = 0$, we have $\mathsf{LIS}(\sigma(B)) \geq \alpha r$. And if $g^{(l)}(\tilde{B}) = 1$, $\mathsf{LIS}(\sigma(B)) \leq (\frac{1}{c} + \frac{1}{l})r$. Here, c and l can be any large constant up to our choice and $\alpha \in (1/2, 1)$ is fixed. For any $\varepsilon > 0$, we can choose c and l such that $(1+\varepsilon)(\frac{1}{c}+\frac{1}{l}) \leq \alpha$. This gives us a reduction from computing $g^{(l)}(\tilde{B})$ to compute a $(1+\varepsilon)$-approximation of $\mathsf{LIS}(\sigma(B))$.

In the t-party game for computing $g^{(l)}(\tilde{B})$, each player holds one column of $\tilde{B}$. Thus, player P_i also holds $C_i(B)$ since $C_i(B)$ is determined by $C_i(\tilde{B})$. If the t players can compute a $(1+\varepsilon)$ approximation of $\sigma(B)$ in the one-way communication model, we can distinguish the case of $g^{(l)}(\tilde{B}) = 0$ and $g^{(l)}(\tilde{B}) = 1$. Thus, any R passes deterministic streaming algorihtm that approximate LIS within a $1+\varepsilon$ factor requires at least $CC_t^{max}(g^{(l)})$. By Lemma 4.4.3, $CC_t^{max}(g^{(l)}) = \Omega(s/R) = \Omega(|\Sigma|/R)$.

□

4.4.2 Longest Non-decreasing Subsequence

We can proof a similar space lower bound for approximating the length of longest non-decreasing subsequence in the streaming model.

Lemma 4.4.5. *For $x \in \Sigma^n$ with $|\Sigma| = O(\sqrt{n})$ and any constant $\varepsilon > 0$, any deterministic algorithm that makes R passes of x and outputs a $(1+\varepsilon)$-approximation of $\mathsf{LNS}(x)$ requires $\Omega(|\Sigma|/R)$ space.*

Proof of Lemma 4.4.5. In this proof, we let the alphabet set $\Sigma = \{1, 2, \dots, (c+1)r\}$. The size of the alphabet is $(c+1)r$. Without loss of generality, we can assume $cr^2 \leq n$.

Let $\tilde{B} \in \{0,1\}^{s \times t}$ be a binary matrix such that for any row of $\tilde{B}$, say $R_i(\tilde{B})$ for $i \in [s]$, either there are at least l 0s between any two 1's, or, $R_i(\tilde{B})$ has at least αt 1's.

Similar to the proof of Lemma 4.4.4, we show a reduction from computing $g((\tilde{B}))$ to approximating the length of LNS. In the following, we set $s = r$ and $t = cr$ for some constant $c > 0$ such that t is an integer.

Let us consider a matrix $B \in \Sigma^{s \times t}$ such that for any $(i,j) \in [s] \times [t]$:

$$B_{i,j} = \begin{cases} i, & \text{if } \tilde{B}_{i,j} = 1, \\ cr + r + 1 - j, & \text{if } \tilde{B}_{i,j} = 0. \end{cases} \tag{4.22}$$

Thus, for all positions (i,j) such that $\tilde{B}_{i,j} = 0$, we know $B_{i,j} > r$. Also, for $1 \le j < j' \le cr$, assume $\tilde{B}_{i,j} = \tilde{B}_{i',j'}$ for some $i, i' \in [r]$, we have $B_{i,j} > B_{i',j'}$. For positions (i,j) such that $\tilde{B}_{i,j} = 1$, we have $B_{i,j} = i$.

Consider the sequence $\sigma(B) = C_1(B) \circ C_2(B) \circ \cdots \circ C_r(B)$ where $C_i(B) = B_{1,i}B_{2,i} \ldots B_{r,i}$ is the concatenation of symbols in the i-th column of matrix B.

We now show that if $g^{(l)}(\tilde{B}) = 0$, $\mathsf{LNS}(\sigma) \le 2r + \frac{cr}{l}$ and if $g(\tilde{B}) = 1$, $\mathsf{LNS}(\sigma) \ge \alpha cr$. [3]

If $g^{(l)}(\tilde{B}) = 0$, consider a longest non-decreasing subsequence of σ denoted by σ'. Than σ' can be divided into two parts σ'^1 and σ'^2 such that σ'^1 consists of symbols from $[r]$ and σ'^2 consists of symbols from $\{r+1, r+2, \ldots, cr\}$. Similar to the proof of Lemma 4.4.4, σ'^1 corresponds to a path on matrix $\tilde{B}$. Since we are concatenating the columns of B, the path can never go left. Each step is either go right at least $l+1$ positions since there are at least l 0's between any two 1's in the same row of $\tilde{B}$, or, go downward to another row. Thus, the total number of steps is at most $\frac{t}{l} + r$ since $\tilde{B}$ has r rows and t columns. For σ'^2, if we restricted B to positions that are in $\{r+1, r+2, \ldots, (c+1)r\}$, symbols in column j of B must be smaller than symbols in column j' if $j < j'$. Thus, the length of σ'^2 must be at most the length of $C_j(B)$ for any $j \in [cr]$, which is at most r. Thus the length of σ is at most $2r + \frac{cr}{l}$.

If $g^{(l)}(\tilde{B}) = 1$, then we know there is some $i \in [r]$ such that row i of $\tilde{B}$ constains at least αcr 1's. We know $B_{i,j} = i$ if $\tilde{B}_{i,j} = 1$. Thus, $R_i(B)$ contains a non-decreasing subseqeunce of length at least αcr. Since R_i is a subsequence of σ. We know $\mathsf{LNS}(\sigma) \ge \alpha cr$.

[3]Here, we assume $n = |\sigma| = cr^2$. If $n > cr^2$, we can repeat each symbol in σ $\frac{n}{cr^2}$ times and show $g^{(l)}(\tilde{B}) = 0$, $\mathsf{LNS}(\sigma) \le \left(2r + \frac{cr}{l}\right)\frac{n}{cr^2}$ and if $g^{(l)}(\tilde{B}) = 1$, $\mathsf{LNS}(\sigma) \ge \left(\alpha cr\right)\frac{n}{cr^2}$. The proof is the same.

For any constant $\varepsilon > 0$, we can pick constants $c, l > 1$ and $\alpha \in (1/2, 1)$ such that $(1+\varepsilon)(2+c/l) \leq \alpha c$. Thus, if we can approximate $\mathsf{LNS}(\sigma)$ to within a $1+\varepsilon$ factor, we can distinguish the case of $g^{(l)}(\tilde{B}) = 0$ and $g^{(l)}(\tilde{B}) = 1$. The lower bound then follows from Lemma 4.4.3.

□

Lemma 4.4.6. *Let $x \in \Sigma^n$ and $\varepsilon > 0$ such that $|\Sigma|^2/\varepsilon = O(n)$. Then any deterministic algorithm that makes constant pass of x and outputs a $(1+\varepsilon)$ approximation of $\mathsf{LNS}(x)$ takes $\Omega(r \log \frac{1}{\varepsilon})$ space.*

Proof. Let the alphabet set $\Sigma = \{a_1, a_2, \dots, a_r\} \cup \{b_1, b_2, \dots, b_r\}$ and assume that

$$b_r < b_{r-1} < \cdots < b_1 < a_1 < a_2 < \cdots < a_r$$

.

We assume $t \geq 2r/\varepsilon$. Since we assume $t = \Omega(r/\varepsilon)$, if $t < 2r/\varepsilon$, we can use less symbols in Σ for our construction and this will not affect the result. Let $l = \Theta(1/\varepsilon)$ that can be divided by 4. For any two symbols a, b, we consider the set $A_{a,b} \in \{a, b\}^l$ such that

$$A_{a,b} = \{a^{\frac{3}{4}l+t} b^{\frac{1}{4}l-t} | 1 \leq t \leq l/4\}$$

We define a function f such that for any $\sigma = a^{\frac{3}{4}l+t} b^{\frac{1}{4}l-t} \in A_{a,b}$, $f(\sigma) = a^{\frac{3}{4}l-t} b^{\frac{1}{4}l+t}$. Thus, for any $\sigma \in A_{a,b}$, the string $\sigma \circ f(\sigma)$ has exactly $\frac{3}{2}l$ a symbols and $\frac{1}{2}l$ b symbols.

We let $\overline{A_{a,b}} = \{f(\sigma) | \sigma \in A_{a,b}\}$. We know $|A_{a,b}| = |\overline{A_{a,b}}| = l/2 = \Theta(1/\varepsilon)$.

Consider an error-correcting code $T_{a,b} \subset S_{a,b}^r$ over alphabet set $S_{a,b}$ with constant rate α and constant distance β. We can pick $\alpha = 1/2$ and $\beta = 1/3$. Then the size of the code $T_{a,b}$ is $|T_{a,b}| = |S_{a,b}|^{\alpha r} = 2^{\Omega(lr)}$. For any code word $\chi = \chi_1 \chi_2 \cdots \chi_r \in T_{a,b}$ where $\chi_i \in S_{a,b}$, we can write

$$\nu(\chi) = (\chi_1, f(\chi_1), \chi_2, f(\chi_2), \dots, \chi_r, f(\chi_r)).$$

Let $W = \{\nu(\chi) | \chi \in T_{a,b}\} \in (A_{a,b} \times \overline{A_{a,b}})^r$. Since the code has constant rate α, the size of W is $(l/4)^{\alpha r}$.

Let $\beta = 1/3$. We can define function $h : (A_{a,b} \times \overline{A_{a,b}})^r \to \{0, 1\}$ such that

$$h(w) = \begin{cases} 0, & \text{if } \forall i \in [r],\ f(w_{2i-1}) = w_{2i}, \\ 1, & \text{if for at least } \beta r \text{ indices } i \in [r],\ w_{2i-1} \circ w_{2i} \\ & \text{ contains more than } \frac{3}{2}l\ a \text{ symbols.} \\ \text{undefined}, & \text{otherwise.} \end{cases} \tag{4.23}$$

Claim 4.4.1. *W is a fooling set for h.*

Proof. Let w and w' be two distinct elements in W. Let

$$w = (w_1, f(w_1), w_2, f(w_2), \ldots, w_r, f(w_r)),$$

$$w' = (w'_1, f(w'_1), w'_2, f(w'_2), \ldots, w'_r, f(w'_r)).$$

By the definition, we know for at least βr positions $i \in [r]$, we have $w_i \neq w'_i$. Also, by the construction of set $S_{a,b}$, if $w_i \neq w'_i$, then one of $w_i \circ f(w'_i)$ and $w'_i \circ f(w_i)$ has more than $\frac{3}{2}l$ a symbols.

v in the span of w and w' if $v = (v_1, \bar{v}_1, \ldots, v_r, \bar{v}_r)$ such that $v_i \in \{w_i, w'_i\}$ and $\bar{v}_i \in \{f(w_i), f(w'_i)\}$. We can find a v in the span of w and w' such that $h(v) = 1$.

□

For $i \in [r]$, we can define A_{a_i,b_i}, $\overline{A_{a_i,b_i}}$, W_i, h_i similarly except the alphabet is $\{a_i, b_i\}$ instead of $\{a, b\}$.

Consider a matrix B of size $r \times 2r$ such that $B_{i,2j-1} \in A_{a_i,b_i}$ and $B_{i,j} \in \overline{A_{a_i,b_i}}$. We define a function g such that

$$g(B) = h_1(R_1(B)) \vee h_2(R_2(B)) \vee \cdots \vee h_r(R_r(B)). \tag{4.24}$$

In the following, we consider a $2r$-party one way game. In the game, player i holds $C_i(B)$. The goal is to compute $g(B)$.

Claim 4.4.2. *The set of all matrix B such that $R_i(B) \in W_i$ for $i \in [r]$ is a fooling set for g.*

Proof. For any two matrix $B_1 \neq B_2$ such that $R_i(B_2), R_i(B_) \in W_i \ \forall\, i \in [r]$. We know $g(B_1) = g(B_2) = 0$. There is some row i such that $R_i(B_1) \neq R_i(B_2)$. We know there is some elements v in the span of $R_i(B_1)$ and $R_i(B_2)$, such that $h_i(v) = 1$ by Claim 4.4.1. Thus, there is some element B' in the span of B_1 and B_2 such that $g(B') = 1$. □

Thus, we get a 2-fooling set for function g in the $2r$-party setting. The size of the fooling set is $|W|^r = (l/4)^{\alpha r^2})$. Thus, $CC_{2r}^{tot}(g) = \log(|W|^r) = \Omega(r^2 \log \frac{1}{\varepsilon})$ and $CC_{2r}^{max}(g) = CC_{2r}^{tot}(g)/2r = \Omega(r \log \frac{1}{\varepsilon})$.

Consider a matrix $\tilde{B}$ of size $r \times 3r$ such that $\tilde{B}$ is obtained by inserting a column of a symbols to B at every third position. Thus,

$$\tilde{B} = \begin{pmatrix} B_{1,1} & B_{1,2} & a_1^{2l} & B_{1,3} & B_{1,4} & a_1^{2l} & \cdots & B_{1,2r-1} & B_{1,2r} & a_1^{2l} \\ B_{2,1} & B_{2,2} & a_2^{2l} & B_{2,3} & B_{2,4} & a_2^{2l} & \cdots & B_{2,2r-1} & B_{2,2r} & a_2^{2l} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ B_{r,1} & B_{r,2} & a_r^{2l} & B_{r,3} & B_{r,4} & a_2^{2l} & \cdots & B_{r,2r-1} & B_{r,2r} & a_r^{2l} \end{pmatrix}_{(r \times 3r)}$$

Let $x = C_1(\tilde{B}) \circ C_2(\tilde{B}) \circ \cdots \circ C_{3r}(\tilde{B})$.

We now show how to reduce computing $g(B)$ to approximating the length of $\mathsf{LNS}(x)$. We claim the following.

Claim 4.4.3. *If $g(B) = 0$, $\mathsf{LNS}(x) \leq \frac{11}{2}rl - 2l$. If $g(B) = 1$, $\mathsf{LNS}(x) \geq \frac{11}{2}rl - 2l + \beta r - 1$.*

Proof. We first divide x into r parts such that

$$x = x^1 \circ x^2 \circ \cdots x^r$$

where $x^i = C_{3i-2}(\tilde{B}) \circ C_{3i-1}(\tilde{B}) \circ C_{3i}(\tilde{B})$. We know $C_{3i-2}(\tilde{B}) = C_{2i-1}(B)$ is the $(2i-1)$-th column of B, $C_{3i-1}(\tilde{B}) = C_{2i}(B)$ is the $2i$-th column of B and $C_{3i}(\tilde{B}) = a_1^{2l} a_2^{2l} \cdots a_r^{2l}$.

If $g(B) = 0$, we show $\mathsf{LNS}(x) = \frac{11}{2}rl - 2l$. Let Σ be a longest non-decreasing subsequence of x. We can divide σ into r parts such that σ^i is a subsequence of x^i. For our analysis, we set $t_0 = 1$. If σ^i is empty or contains no a symbols, let $t_i = t_{i-1}$. Otherwise, we let t_i be the largest integer such that a_{t_i} appeared in σ^i.We have

$$1 = t_0 \le t_1 \le t_2 \le \cdots \le t_r \le r$$

We now show that, for any i, the length of σ^i is at most $\frac{3}{2}l + 2(t_i - t_{i-1} + 1)l$. To see this, if $t_i = t_{i-1}$, x^i to σ. Since $g(B) = 0$, there are exactly $\frac{3}{2}l + 2l$ a_{t_i} symbols in x^i. Thus $|\sigma^i| \le \frac{3}{2}l + 2l$.

If $t_i > t_{i-1}$, if there is some a_t symbol in the $C_{3i-2}(\tilde{B}) \circ C_{3i-1}(\tilde{B})$ included in σ^i for some $t_{i-1} < t \le t_i$. Then σ^i can not include $a_{t'}$ symbols in $C_{3i}(\tilde{B})$ for all $t_{i-1} \le t' < t$. Also for any $t_{i-1} \le t' < t$, the number of $a_{t'}$ symbols in $C_{3i-2}(\tilde{B}) \circ C_{3i-1}(\tilde{B})$ is $\frac{3}{2}l$ but the number in $C_{3i}(\tilde{B})$ is $2l$. Thus, the optimal strategy it to pick the $\frac{3}{2}l$ $a_{t_{i-1}}$ symbols in $C_{3i-2}(\tilde{B}) \circ C_{3i-1}(\tilde{B})$ and then add $2(t_i - t_{i-1} + 1)l$ symbols ($2l$ a_t's for $t_{i-1} \le t \le t_i$) in $C_{3i}(\tilde{B})$ to σ^i.

In total, length of σ is at most

$$\sum_{i=1}^{r} \sigma^i \le \frac{11}{2}rl - 2l$$

Thus, if $g(B) = 0$, we know $\mathsf{LNS}(x) \le \frac{11}{2}rl - 2l$.

If $g(B) = 1$, that means there is some row i of B such that for at least βr positions $j \in [r]$, the number of a_i symbols in $B_{i,2j-1} \circ B_{i,2j}$ is at least $\frac{3}{2}l + 1$.

We now build a non-decreasing subsequence σ of x with length at least $\frac{11}{2}rl - 2l + \beta r - 1$. We set σ to be empty initially. There are at least $\frac{3}{2}l$ a_1 symbols in $B_{1,1} \circ B_{1,2}$. We add all of them to σ. Then, we add all the a_t symbols for $1 \le t \le i$ in $C_3\tilde{B}$ to σ. This adds $2il$ symbols to σ

We consider the string

$$\tilde{x} = B_{i,3} \circ B_{i,4} \circ \circ a_i^{2l} \circ \cdots \circ B_{i,2r-1} \circ B_{i,2r} \circ a_i^{2l}.$$

It is a subsequence of $x^2 \circ \cdots x^r$. There are $(r-1)(\frac{3}{2}l + 2l + \beta r - 1)$ a_i symbols in $\tilde{x}$. This is because for at least $\beta r - 1$ positions $j \in \{2, 3, \ldots, r\}$, we know $B_{i,2j-1} \circ B_{i,2j}$ contains more than $\frac{3}{2}l$ a_i symbols. For the rest of the positions, we know $B_{i,2j-1} \circ B_{i,2j}$ contains $\frac{3}{2}l$ a_i symbols.

Finally, we add all the a_t symbols for $i < t \leq n$ in $C_{3r}\tilde{B}$ to σ. This adds another $2(r-i)l$ symbols to σ. The sequence σ has length at least $\frac{11}{2}rl - 2l + \beta r - 1$. □

Assume $l = \lambda/\varepsilon$ where λ is some constant. Let $\varepsilon' = \frac{\beta}{10\lambda}\varepsilon = \Theta(\varepsilon)$. If we can give a $1 + \varepsilon'$ approximation of $\mathsf{LNS}(x)$, we can distinguish $g(B) = 0$ and $g(B) = 1$.

By the fact that $CC_{2r}^{max}(g) = \Omega(r \log \frac{1}{\varepsilon})$, any deterministic streaming algorithm with constant passes of x that approximate $\mathsf{LNS}(x)$ within a $(1 + \varepsilon')$ factor requires $\Omega(r \log \frac{1}{\varepsilon})$ space. □

4.4.3 Longest Non-decreasing Subsequence with Threshold

We now consider a variant of LNS problem we call longest non-decreasing subsequence with threshold (LNST). In this section, we assume the alphabet is $\Sigma = \{0, 1, 2, \ldots, r\}$. In this problem, we are given a sequence $x \in \Sigma^n$ and a threshold $t \in [n]$, the longest non-decreasing subsequence with threshold t is the longest non-decreasing subsequence of x such that each symbol appeared in it is repeated at most t times. We denote the length of such a subsequence by $\mathsf{LNST}(x, t)$.

Lemma 4.4.7. *Let $x \in \Sigma^n$ and $|\Sigma| = r \leq \sqrt{n}$) and $\varepsilon > 0$ be any constant. Let $t \geq n/r$. Then any R-pass deterministic algorithm a $1+\varepsilon$ approximation of $\mathsf{LNST}(x,t)$ takes $\Omega(r/R)$ space.*

Proof. In the proof of Lemma 4.4.5, each symbol in sequence σ appeared no more than $\max(r, n/r)$ times. If $r \leq \sqrt{n}$ and $t \geq n/r$, we have $\mathsf{LNS}(\sigma) = \mathsf{LNST}(\sigma, t)$. The lower bound follows from our lower bound for LNS in Lemma 4.4.5.

□

Lemma 4.4.8. *Let $x \in \Sigma^n$ and $|\Sigma| = r$ and $\varepsilon > 0$ be any constant. Let t be some constant. Then any R pass deterministic algorithm outputs a $1+\varepsilon$ approximation of $\mathsf{LNST}(x, t)$ takes $\Omega(\min(\sqrt{n}, r))$ space.*

Proof. Let $\sigma \in \Sigma^{n/t}$. If we repeat every symbol in σ t times, we get a string $\sigma' \in \Sigma$. Then, $\mathsf{LIS}(x) = \frac{1}{t}\mathsf{LNST}(x, t)$. When t is a constant, the lower bound follows from lower bounds for LIS in Lemma 4.4.4. □

Lemma 4.4.9. *Let $x \in \Sigma^n$. Assume $|\Sigma| = r$ and $\varepsilon > 0$ such that $r^2/\varepsilon = O(n)$. Let $t = \Theta(r/\varepsilon)$. Then any R-pass deterministic algorithm that outputs a $1+\varepsilon$ approximation of $\mathsf{LNST}(x, t)$ takes $\Omega(r \log 1/\varepsilon$ space.*

Proof. In the proof of Lemma 4.4.6, we considered strings x where each symbol is appeared at most $4rl$ times where $l = 1/\varepsilon$. We $t = 4rl = \Theta(r/\varepsilon)$. Thus, $\mathsf{LNST}(x, t) = \mathsf{LNS}(x)$. The lower bound follows from Lemma 4.4.6.

□

Lemma 4.4.10. *Assume $x \in \Sigma^n$, $\varepsilon > 0$, and $\frac{|\Sigma|}{\varepsilon} = O(n)$. Let $t = \Theta(1/\varepsilon)$, any R-pass deterministic algorithm that outputs an $1+\varepsilon$ approximation of $\mathsf{LNST}(x, t)$ requires $\Omega(\frac{\sqrt{|\Sigma|}}{\varepsilon})$ space.*

Proof. The lower bound is achieved using the same construction in the proof of Theorem 4.3 with some modifications. In Section 4.3.1, for any n, we build a fooling set $S_{a,b} \subset \{a, b\}^{n/2}$ and $\overline{S_{a,b}} \subseteq \{a, b\}^{n/2+5}$ (we used the notation S instead of $S_{a,b}$ in Section 4.3.1) such that $\overline{S_{a,b}} = \{f(x)|x \in S_{a,b}\}$ where the function f simply delete the first 10 b's in x. We prove Lemma 4.3.1 and Claim 4.3.1. We modify the construction of $S_{a,b}$ and $\overline{S_{a,b}}$ with three symbols a, b, c. The modification is to replace every a

symbols in the strings in $\overline{S_{a,b}}$ with c symbols. This gives us a new set $\overline{S_{b,c}} \subset \{b,c\}^{n/2-5}$. Thus, the function f now becomes, on input $x \in S_{a,b}$, first remove 10 b's and then replace all a symbols with c.

Let $y = a^{n/3}b^{n/3}c^{n/3}$. We can show that for every $x \in S_{a,b}$,

$$\mathsf{LCS}(x \circ f(x), y) = \frac{n}{2} + 5.$$

Also, for any two distinct $x^1, x^2 \in S_{a,b}$,

$$\max\{\mathsf{LCS}(x^1 \circ f(x^2), y), \mathsf{LCS}(x^2 \circ f(x^1), y)\} > \frac{n}{2} + 5.$$

The proof is the same as the proof of Lemma 4.3.1.

We now modify the construction in the proof of Theorem 4.3. Since $t = \Theta(1/\varepsilon)$, we choose n_ε such that $n_\varepsilon/3 = t$. More specifically, for the matrix $\bar{B}$ (see equation 4.11), we do the following modification. For any $i, j \in [r]$, $y^{i,2j-1} \circ y^{i,2j} = a_i^{n_\varepsilon/3} b_i^{n_\varepsilon/3} a_i^{n_\varepsilon/3}$. We replace $y^{i,2j-1} \circ y^{i,2j}$ with string $d_{i,j}^{n_\varepsilon/3} e_{i,j}^{n_\varepsilon/3} f_{i,j}^{n_\varepsilon/3}$. Here, $d_{i,j}, e_{i,j}, f_{i,j}$ are three symbols in Σ such that $d_{i,j} < e_{i,j} < f_{i,j}$. In the matrix $\tilde{B}$ (see equation 4.10), for $B_{i,2j-1}$, we replace every a_i symbols with $d_{i,j}$ and b_i symbols with $e_{i,j}$. For $B_{i,2j}$, we place a_i symbols with $f_{i,j}$ and b_i symbols with $e_{i,j}$.

We also replace the $c_j^{n_\varepsilon}$ block in the i-th row of both $\bar{B}$ and $\tilde{B}$ with $c_{i,j,1}^{n_\varepsilon/3} c_{i,j,2}^{n_\varepsilon/3} c_{i,j,2}^{n_\varepsilon/3}$. Here, $c_{i,j,1}, c_{i,j,2}.c_{i,j,3}$ are three different symbols in Σ such that $c_{i,j,1} < c_{i,j,2} < c_{i,j,3}$.

y is the concatenation of rows of $\bar{B}$. We require that symbols appeared earlier in y are smaller. Since y is the concatenation of all symbols that appeared in x and each symbol in y repeated $t = n_\varepsilon$ times. After the symbol replacement, we have $\mathsf{LCS}(x, y) = \mathsf{LNST}(x, t)$. Also notice that the alphabet size is now $O(r^2)$ instead of r in the proof of Theorem 4.3. The $\Omega(\frac{|\Sigma|}{\varepsilon})$ space lower bound then follows from a similar analysis in the proof of Theorem 4.3.

□

Theorem 4.4 is a direct result of Lemma 4.4.7, Lemma 4.4.8, Lemma 4.4.9 and Lemma 4.4.10.

Algorithm 12: Approximate the length of the LNST

```
Data: An online string $x \in \Sigma^n$ where $\Sigma = [r]$.
1: Let $D = \{0, \frac{\varepsilon}{r}t, 2\frac{\varepsilon}{r}t, \dots, t\}$        ▷ if $r/\varepsilon \geq t$, $D = [t] \cup \{0\}$.
2: $S \leftarrow \emptyset$.
3: for all $d = (d_1, \dots, d_r) \in \mathcal{D}^r$ do
                                  ▷ try every $d = (d_1, \dots, d_r) \in \mathcal{D}^r$ in parallel
4:     Let $\sigma(d) = 1^{d_1}2^{d_2}\cdots r^{d_r}$.
5:     If $\sigma(d)$ is a subsequence of $x$, add $d$ to $S$.
6: In parallel, compute $\mathsf{LNS}(x)$ exactly with an additional $\tilde{O}(r)$ bits of space.
7: if $\mathsf{LNS}(x) \leq t$ then
8:     return $\mathsf{LNS}(x)$.
9: else
10:    return $\max_{d \in S}\left(\sum_{i\in[r]} d_i\right)$.
```

A Simple Upper Bound for LNST

Proof of Theorem 4.5. We assume $\Sigma = [r]$ and the input is a string $x \in \Sigma^n$. Let σ be a longest non-decreasing subsequence of x with threshold t and we can write $\sigma = 1^{n_1}2^{n_2}\cdots r^{n_r}$ where n_i is the number of times symbol i repeated and $0 \leq n_i \leq t$.

We let D be the set $\{0, \frac{\varepsilon}{r}t, 2\frac{\varepsilon}{r}t, \dots, t\}$. Thus, if $r/\varepsilon \geq t$, $D = [t] \cup \{0\}$. Consider the set $\mathcal{D}$ such that $\mathcal{D} = D^r$. Thus, $|\mathcal{D}| = \left(\min(t+1, r/\varepsilon + 1)\right)^r$. For convenience, we let $f(d) = \sum_{i=1}^r d_i$. We initialize the set S to be an empty set. For each $d \in \mathcal{D}$, run in parallel, we check is $\sigma(d) = 1^{d_1}2^{d_2}\cdots r^{d_r}$ is a subsequence of x. If $\sigma(d)$ is a subsequence of x, add d to S.

Meanwhile, we also compute $\mathsf{LNS}(x)$ exactly with an additional $\tilde{O}(r)$ bits of space. If $\mathsf{LNS} < t$, we output $\mathsf{LNS}(x)$. Otherwise, we output $\max_{d\in S}\left(\sum_{i\in[r]} d_i\right)$.

We now show the output is a $(1-\varepsilon)$-approximation of $\mathsf{LNST}(x,t)$. Let d'_i be the largest element in $\mathcal{D}$ that is no larger than n_i for $i \in [r]$ and $d' = (d'_1, d'_2, \dots, d'_r)$.

If $t \leq |\sigma|$, we have $0 \leq n_i - d'_i \leq \varepsilon/rt$ and

$$|\sigma| - f(d') \leq \sum_{i=1}^{r}(n_i - d'_i) \leq \varepsilon t \leq \varepsilon|\sigma|$$

since we assume $t \leq |\sigma|$. Thus, $f(d') \geq (1-\varepsilon)|\sigma|$. Note that $1^{d'_1}2^{d'_2}\cdots r^{d'_r}$ is a subsequence of σ and thus also a subsequence of x. Thus, we add d' to the set S. That means, the final output will be at least $f(d')$. Denote the final output by l, we have

$$l \geq f(d') \geq (1-\varepsilon)|\sigma|.$$

On the otherhand, the the output is $f(d)$ for some $d = (d_1, \ldots, d_r)$, we find a subsequence $1^{d_1}2^{d_2}\cdots r^{d_r}$ of x and thus $f(d) \leq |\sigma|$. We know

$$l = \max_{d \in S} f(d) \leq |\sigma|.$$

If $t \geq \mathsf{LNS}(x)$, no symbol in σ is repeated more than $t-1$ times. Thus, $\mathsf{LNST}(x,t) = \mathsf{LNS}(x)$. Thus, we output $\mathsf{LNS}(x)$. Notice that if $\mathsf{LNS}(x) > t$, either some symbol in the longest non-decreasing subsequence is repeated more than t times, or $\mathsf{LNST}(x,t) = \mathsf{LNS}(x)$. In either case, we have $t \leq |\sigma|$ and $\max_{d\in S}\left(\sum_{i\in[r]} d_i\right)$ is a $1-\varepsilon$ approximation of $\mathsf{LNS}(x)$.

□

4.5 Open Problems

We list some of the interesting open problems below.

1. Our lower bounds for ED and the techniques there do not apply to the case where x is a permutation of y (i.e., the Ulam distance). Is there a way to get similar bounds for Ulam distance, or is there a better algorithm for Ulam distance in the asymmetric streaming model? Furthermore, can we get better bounds for small alphabets?

2. Our lower bounds for approximating ED (even over a large alphabet) is not strong and does not match the upper bounds. Can we get a better lower bound or a better algorithm?

3. For small alphabets, it is not clear whether our lower bounds for approximating LCS are tight. Can we get better lower bounds or better approximation algorithms in this case? In particular, is the correct dependence on $1/\varepsilon$ linear as in our bounds, or is there a threshold phenomenon? We note that a natural idea towards a better lower bound is to use a more "random" string y, but then the analysis of LCS becomes quite tricky.

4. How hard is LNST? As a natural generalization of LIS, LNS and a special case of LCS (when $|\Sigma|t \leq n$), does LNST fully capture the hardness of LCS? We note that our best bound for $1+\varepsilon$ approximation of LCS is $|\Sigma|/\varepsilon$ while for LNST it is just $\sqrt{|\Sigma|/\varepsilon}$. Can we get a better bound for LNST? Can we provide a complete characterization of the space complexity of LNST for all t, including designing better streaming algorithms for LNST?

Chapter 5

Locally Decodable Codes for Edit Errors

5.1 Introduction

In this chapter, we present our results about locally decodable codes.

5.1.1 Main Results

We give two sets of results regarding LDCs. In the first set of results, we establish exponential lower bound for LDCs correcting edit errors. In the second set of results, we introduce the notion of LDCs with randomized encoding and show that with randomized encoding, we can achieve significantly better rate-query tradeoffs for both Hamming and edit errors.

Lower Bounds

Our lower bounds are for LDCs that can tolerate edit errors (Insdel LDCs). The first result shows that 2-query linear Insdel LDCs do not exist, which means that no matter how long the codeword is, the LDC can only encode a constant number of message bits.

Theorem 5.1. *For any $(2, \delta, \varepsilon)$ linear or affine Insdel LDC $C : \{0,1\}^n \to \{0,1\}^m$, we have $n = O_{\delta,\varepsilon}(1)$.*

More generally, we show an exponential lower bound for general 2-query Insdel LDCs.

Theorem 5.2. *For any $(2, \delta, \varepsilon)$ Insdel LDC $C : \{0,1\}^n \to \{0,1\}^m$, we have $m = \exp(\Omega_{\delta,\varepsilon}(n))$.*

We remark that, as previously mentioned, the lower bound for 2-query Hamming LDCs from [57] also holds for 2-query Insdel LDCs. However, that proof uses sophisticated quantum arguments, and an important quest in the area has been providing non-quantum proofs for the same result. Indeed, the proof from [57] was adapted to classical arguments by [73], but the arguments still retained a strong quantum-style flavor. Our arguments here do not resemble those proofs and are purely classical. Furthermore, in contrast to the lower bounds from [57, 73], our lower bounds in Theorems 5.1 and 5.2 extend to the private-key setting where the encoder and decoder share private randomness.

We prove the following general bound for $q \geq 3$ queries.

Theorem 5.3. *For any non-adaptive (q, δ, ε) Insdel LDC $C\colon \{0,1\}^n \to \{0,1\}^m$ with $q \geq 3$, we have the following bounds.*

$$m = \begin{cases} \exp(\Omega_{\delta,\varepsilon}(\sqrt{n})) \textit{ for } q = 3\textit{; and} \\ \exp\left(\Omega\left(\frac{\delta}{\ln^2(q/\varepsilon)} \cdot \left(\varepsilon^3 n\right)^{1/(2q-4)}\right)\right) \textit{ for } q \geq 4. \end{cases}$$

As a comparison, for general Hamming LDCs the best known lower bounds for $q \geq 3$ in [60] give $m = \Omega(n^2/\log n)$ for $q = 3$, and $m = \Omega(n^{1+1/\lceil (q-1)/2 \rceil}/\log n)$ for $q > 3$. Thus, in the constant-query regime, the bounds from Theorem 5.3 are essentially exponential in the existing bounds for Hamming LDCs. Moreover, these bounds also give a separation between constant-query Hamming LDCs, which can have length $\exp(n^{o(1)})$, and constant-query Insdel LDCs.

Lower Bounds for Adaptive Decoders It is well-known [46] that a (q, δ, ε) adaptive Hamming LDC can be converted into a non-adaptive $(\frac{|\Sigma|^q - 1}{|\Sigma| - 1}, \delta, \varepsilon)$ Hamming LDC, and also into a non-adaptive $(q, \delta, \varepsilon/|\Sigma|^{q-1})$ Hamming LDC, and hence lower bounds for non-adaptive decoders imply lower bounds for adaptive decoders, with the respective loss in parameters. It is easy to verify that the same reduction works for Insdel LDCs.[1] In particular our lower bounds imply the respective lower bounds for adaptive decoders.

Corollary 5.1.1. *For any (possibly adaptive) (q, δ, ε) Insdel LDC $C\colon \{0,1\}^n \to \{0,1\}^m$ with $q \geq 3$, we have the following bounds for arbitrary adversarial channels*

$$m = \begin{cases} \exp(\Omega_{\delta,\varepsilon}(\sqrt{n})) \text{ for } q = 3\text{; and} \\ \exp\left(\Omega\left(\frac{\delta}{(q+\ln(q/\varepsilon))^2} \cdot \left(\varepsilon^3 n\right)^{1/(2q-4)}\right)\right) \text{ for } q \geq 4. \end{cases}$$

Corollary 5.1.1 is obtained by plugging $\epsilon' = \epsilon/2^{q-1}$ into Theorem 5.3 and applying the average case reduction from a (q, δ, ϵ) (adaptive) Insdel LDC to a $(q, \delta, \epsilon/2^{q-1})$ (non-adaptive) Insdel LDC [46]. Corollary 5.1.1 also implies lower bounds in regimes where q is slightly super-constant (but $o(\log n)$).

Corollary 5.1.2. *For any (possibly adaptive) (q, δ, ε) Insdel LDC $C\colon \{0,1\}^n \to \{0,1\}^m$, the following bounds hold.*

- *If $q = O(\log\log n)$, then $m = \exp\Big(\exp(\Omega_{\delta,\varepsilon}(\log n/\log\log n))\Big)$.*
- *If $q = \log n/(2c\log\log n)$ for some $c > 3$, then $m = \exp(\Omega(\log^{c-2} n))$. In turn, if $m = \mathrm{poly}(n)$, then $q = \Omega(\log n/\log\log n)$.*

[1]For example, our non-adaptive decoder can pick $r_1, \ldots, r_{q-1} \in \Sigma$ randomly and simulate the adaptive (q, δ, ϵ)-decoder responding to the first $q-1$ queries with $r_1, \ldots, r_{q-1}$. This allows the non-adaptive decoder to extract a set $(j_1, \ldots, j_q)$ of queries representing the set of queries that the adaptive decoder would have asked given the first $q-1$ responses. The queries $(j_1, \ldots, j_q)$ can then be asked non-adaptively to obtain $y[j_1], \ldots, y[j_q]$. With probability $|\Sigma|^{-q+1}$ we will have $y[j_i] = r_i$ for each $i \leq q-1$ and we can finish simulating the adaptive decoder to obtain a prediction x_i which will be correct with probability at least $\frac{1}{2} + \varepsilon$. Otherwise, our non-adaptive decoder randomly guesses the output bit x_i. Thus, the non-adaptive decoder is successful with probability at least $\frac{1}{2} + \epsilon|\Sigma|^{-q+1}$.

We remark that the lower bound for $q = O(\log\log n)$ queries is even larger than the Hamming LDC upper bound of $\exp(\exp((\log n)^{1/t}(\log\log n)^{1-1/t}))$ due to [61, 63, 64] for $q = 2^t$ being a constant number of queries.

Furthermore, we get a super-polynomial lower bound even if $q = \log n/(8\log\log n)$. Thus to get any polynomial length Insdel LDC one needs $q = \Omega(\log n/\log\log n)$. This can be compared to the Insdel LDC constructions in [97, 98], which give $m = o(n^2)$ with $q = (\log n)^C$ for some $C > 2$ (or to the private-key Insdel LDC construction in [99, 100] which gives constant rate $m = \Theta(n)$ and $q = (\log n)^C$ for some $C > 2$). Both the lower bound and the upper bound are for an adaptive Insdel LDC, so our lower bound on the query complexity almost matches the upper bound for polynomial length Insdel LDCs. This also implies that there is a "phase transition" phenomenon in the $q = \text{polylog}(n)$ regime, where the length of the Insdel LDC transits from super-polynomial to polynomial.

LDCs with Randomized Encoding

We first restate the definition of LDCs with randomized encoding below.

Definition 1.3.2. *[LDC with a fixed failure probability]*

An $(m, n, \delta, q, \varepsilon)$ LDC with randomized encoding consists of a pair of randomized functions $\{\mathsf{Enc}, \mathsf{Dec}\}$, such that:

- $\mathsf{Enc} : \{0,1\}^n \to \{0,1\}^m$ *is the encoding function. For every message $x \in \{0,1\}^k$, $y = \mathsf{Enc}(x) \in \{0,1\}^n$ is the corresponding codeword.*

- $\mathsf{Dec} : [n] \times \{0,1\}^* \to \{0,1\}$ *is the decoding function. If the adversary adds at most δn errors to the codeword, then for every $i \in [n]$, every $y \in \{0,1\}^*$ which is a corrupted codeword,*

$$\Pr[\mathsf{Dec}(i, y) = x_i] \geq 1 - \varepsilon,$$

where the probability is taken over the randomness of both Enc *and* Dec.

- Dec *makes at most q queries to y.*

We also provide a variant that has flexible failure probabilities, which is restated below.

Definition 1.3.3. *[LDC with flexible failure probabilities]*

An (m, n, δ) LDC with randomized encoding and query complexity function $q : \mathbb{N} \times [0,1] \to \mathbb{N}$, consists of a pair of randomized algorithms $\{\mathsf{Enc}, \mathsf{Dec}\}$, such that:

- $\mathsf{Enc} : \{0,1\}^n \to \{0,1\}^m$ *is the encoding function. For every message $x \in \{0,1\}^n$, $y = \mathsf{Enc}(x) \in \{0,1\}^m$ is the corresponding codeword.*

- $\mathsf{Dec} : [n] \times \{0,1\}^* \to \{0,1\}$ *is the decoding function. If the adversary adds at most δn errors to the codeword, then for every $i \in [n]$, every $y \in \{0,1\}^*$ which is a corrupted codeword, and every $\varepsilon \in (0,1]$,*

  $$\Pr[\mathsf{Dec}(i, y) = x_i] \geq 1 - \varepsilon,$$

  while Dec makes at most $q = q(m, \varepsilon)$ queries to y. The probability is taken over the randomness of both Enc and Dec.

Recall that we study constructions in the following two models:

Shared randomness In this model, the encoder and the decoder share a private uniform random string. Thus, the adversary does not know the randomness used by the encoder; but he can add arbitrary errors to the codeword, including looking at the codeword first and then adaptively add errors.

Oblivious channel In this model, the encoder and the decoder do *not* share any randomness. However, the communication channel is oblivious, in the sense that the adversary can add any error pattern *non adaptively*, i.e., without looking at the codeword first.

In this thesis, we provide constructions with rate-query tradeoff better than standard locally decodable codes. We have the following theorems. The first one deals with a fixed decoding failure probability.

Theorem 5.4. *There exists a constant $\delta > 0$ such that for every $n \in \mathbb{N}$ and any $\varepsilon \in (0,1)$, there is an efficient construction of $(m, n, \delta, q, \varepsilon)$ LDC with randomized encoding that can tolerate edit errors. It has codeword length $m = O(n)$ and the query complexity is $q = \operatorname{polylog} n \log \frac{1}{\varepsilon}$ for both the shared randomness model and the oblivious channel model.*

We can compare our theorem to standard locally decodable codes. Achieving $q = O(\operatorname{polylog} n \log \frac{1}{\varepsilon})$ is equivalent to achieving query complexity $\operatorname{polylog} n$ for success probability 2/3. For standard LDCs for edit errors, the best known construction is by applying the compiler from [97] to the construction to Hamming errors by [69], which has query complexity $2^{O(\sqrt{\log n \log\log n})}$. For constructions with query complexity $\operatorname{polylog} n$, the best construction is by applying the compiler from [97] to Reed-Muller codes (see [62] for a survey) which has codeword length $m = O(n^{1+c})$ where c is a constant depending on the query complexity.

The next theorem deals with a flexible decoding failure probability. We note that one way to achieve this is to repeat the encoding several times independently, and send all the obtained codewords together. The decoding will then decode from each codeword and take a majority vote. However, this approach can decrease the rate of the code dramatically. For example, if one wishes to reduce the failure probability from a constant to $2^{-\Omega(n)}$, then one needs to repeat the encoding for $\Omega(n)$ times and the rate of the code decreases by a factor of $1/n$. In this work we use a different construction that can achieve a much better rate.

Theorem 5.5. *There exists a constant $\delta > 0$ such that for every $n \in \mathbb{N}$ there is an efficient construction of (m, n, δ) LDC with randomized encoding and flexible failure* *probability that can tolerate edit errors. It has codeword length $m = O(n \log n)$ and for failure probability $\varepsilon \in (0,1)$, the query complexity is $q = \mathrm{polylog}\, n \log \frac{1}{\varepsilon}$ for both the shared randomness model and the oblivious channel model.*

Remark 5.6. *In the original paper [100], the authors also give constructions for Hamming errors. Specifically, for fixed failure probability, [100] gives constructions with constant rate with query complexity $q = O(\log(1/\varepsilon))$ for Hamming errors in the shared randomness model, and query complexity $q = O(\log n \log(1/\varepsilon))$ for Hamming errors in the oblivious channel model. For flexible failure probability, [100] gives constructions with codeword length $m = O(n \log n)$ with query complexity $q = O(\log n \log(1/\varepsilon))$ for Hamming errors in the shared randomness model, and query complexity $q = O(\log^2 n \log(1/\varepsilon))$ for Hamming errors in the oblivious channel model. Since in this thesis we are focusing on edit errors, we omit these results here.*

5.1.2 Overview of Techniques

Here we give an informal overview of the key ideas and techniques used in our proofs of the lower bounds and the constructions.

5.1.2.1 Lower Bounds

We start with the lower bounds. In this section, we always assume a non-adaptive decoder in the following discussion.

Prior strategies for Hamming LDC lower bounds We start by discussing the proof strategies in lower bounds for Hamming LDCs. Essentially all such proofs[2] begin by observing that the code needs to be *smooth* in the sense that for any target message bit, the decoder cannot query a specific index with very high probability. Using this property, one can show that if we represent the queries used by the decoder as edges

[2]Except the proof in [65] which gives a lower bound for 3-query Hamming LDC in a special range of parameters.

in a hypergraph with m vertices, then for any target message bit the hypergraph contains a matching of size $\Omega(m/q)$. The key idea in the proof is now to analyze this matching, where one uses various tools such as (quantum) information theory [46, 57, 60], matrix hypercontractivity [73], combinatorial arguments [46, 70], and reductions from q-query to 2-query [60, 72].

For our proofs, however, the matching turns out to be not the right object to look at. Indeed, by simply analyzing the matching it is hard to prove any strong lower bounds for $q \geq 3$, as evidenced by the lack of progress for Hamming LDCs. Intuitively, a matching does not capture the essence of Insdel errors (e.g., position shifts), which are strictly more general and powerful than Hamming errors. Therefore, we instead need to look at a different object.

The Good queries For a q-query Insdel LDC, the correct object turns out to be the set of all *good* q-tuples in the codeword that are potentially useful for decoding a target message bit. When we view the bits in the codeword as functions of the message, we define a q-tuple to be good for the i'th message bit if there exists a Boolean function $f : \{0,1\}^q \to \{0,1\}$ which can predict the i'th message bit with a non-trivial advantage (e.g., with probability at least $1/2 + \varepsilon/4$, see Definition 5.1.1), using these q bits. It is a straightforward application of information theory (e.g., Theorem 2 in [46]) that any q-tuple cannot be good for too many message bits. Therefore, intuitively, if we can show that any message bit requires a lot of good tuples to decode, then we can conclude that there must be many tuples and thus the codeword must be long. In the extreme case, if we can show that any message bit requires a *constant fraction* of all tuples to decode, then we can conclude that there can be at most a constant number of message bits, regardless of the length of the codeword.

Towards this end, we consider the effect of Insdels on the tuples. Suppose the decoder originally queries some q-tuple A. After some Insdels (e.g., deletions) the positions of the tuples will change, and the actual tuple the decoder queries using A now may correspond to some other tuple B in the original codeword. B may not be a good tuple, in which case it's not useful for decoding the message bit. However, since the decoder always succeeds with probability $1/2 + \varepsilon$ when the number of errors is bounded, the decoder should still hit good tuples with a decent probability (e.g., $3\varepsilon/2$). Intuitively, this already implies in some sense that there should be many good tuples, except that this depends on the decoder's probability distribution. For example, if the decoder queries one tuple with probability 1, then for any fixed error pattern one just needs to make sure that one specific tuple is good.

To leverage the above point, we turn to a probabilistic analysis and use random errors. Specifically, we carefully design a probability distribution on the Insdel errors. For any q-tuple A, this distribution also induces another probability distribution for the q-tuple B which A corresponds to in the original codeword. The key ingredient in all our proofs is to design the error distribution such that it ensures certain nice properties of the induced distribution of any q-tuple, which will allow us to establish our bounds. This can be viewed as a conceptual contribution of our work, as we have reduced the problem of proving lower bounds of Insdel LDCs to the problem of designing appropriate error distributions.

Designing the Insdel error distribution What is the best Insdel error distribution for our proof? It turns out the ideal case for the induced distribution of a q-tuple is the uniform distribution. Indeed, the hitting property discussed above implies that for any message bit, there is at least one q-tuple in the support of the decoder's queries which would still be good with constant probability under the induced distribution. If we can design an error distribution such that for *any* q-tuple, the induced distribution is the uniform distribution on all q-tuples, this means that for any message bit, there are at least a constant fraction of all q-tuples that are good for this bit, which would in turn imply that there can be at most a constant number of message bits.

However, it appears hard to design an error distribution with the above property even for $q = 2$, since we have a bound on the total number of errors allowed, and errors allocated to one tuple will affect the number of errors available for other tuples. Instead, our goal is to design the error distribution such that the induced distribution of any q-tuple is as "close" to the uniform distribution as possible. We first illustrate our ideas for the case of $q = 2$.

The case of $q = 2$ A simple idea is to start with a random number (up to $\Omega(m)$) of deletions at the beginning of the codeword, we call this deletion *type* **1**. This results in a random shift of any pair of indices. However, a crucial observation is that the *distance* between any pair of indices stays the same (for a pair of indices $i, j \in [m]$, their distance is $|i - j|$), which makes the induced distribution far from being uniform. Indeed, under such error patterns the Hadamard code seems to be a good candidate for Insdel LDC. This is because any codeword bit of the Hadamard code is the inner product of a vector $v \in \{0, 1\}^n$ with the message, and to decode the i'th message bit the decoder queries a pair of inner products for v and $v + e_i$ (e_i is the i'th standard basis vector) where v is a uniform vector. If we arrange the codeword bits in the natural lexicographical order according to v, then all pairs used in queries for the i'th message bit have a fixed distance of 2^{i-1}. In fact we show in the appendix that a simple variant of the Hadamard code does give a LDC under deletion type **1**. However, our Theorem 5.1 implies that it is not an Insdel LDC in general. The point here is that we need a different operation to change the distance of any pair, which is a phenomenon unique to Insdel LDC and never happens in Hamming LDC.

To achieve this, we introduce *random deletions* of each message bit on top of the previous operation. Specifically, imagine that we fix a constant $p < \delta$ and delete each bit of the codeword independently with probability p. Under this error distribution, any pair of queries with distance d will correspond to a pair with distance $\frac{d}{1-p}$ in expectation (since we expect to delete p fraction of bits in any interval). However, the independent deletions lead to a concentration around the mean. Thus the probability of any distance around $\frac{d}{1-p}$ is $\Theta(\frac{1}{\sqrt{d}})$ and the distribution resembles that of a binomial distribution (it is called a negative binomial distribution), which is not flat enough compared to the uniform distribution. Therefore, we add another twist by first picking the parameter p uniformly from an interval (e.g., $[\frac{\delta}{8}, \frac{\delta}{4}]$) and then delete each bit of the codeword independently with probability p. We call this deletion *type* **2**. Somewhat magically, the compound distribution now effectively "flattens" the original distribution, and we can show that the probability mass of any distance is now $O(\frac{1}{d})$. Intuitively, this is because the distance in the induced distribution is now roughly equally likely to appear in the interval $[\frac{d}{1-\delta/8}, \frac{d}{1-\delta/4}]$. Combined with the deletions at the beginning, we can conclude the following two properties for any pair with original distance d in the induced distribution: (1) The probability mass of any element in the support is $O(\frac{1}{md})$, and (2) With high probability, the corresponding pair will have distance in $[d, cd]$ for some constant $c = c(\delta, \varepsilon)$ (See Lemma 5.2.1 for the formal statement).

While this is not exactly the uniform distribution, it is already enough to establish non-trivial bounds. To do this, we divide all pairs of queries into $O(\log m)$ intervals based on their distances, where the j'th interval P_j consists of all pairs with distance in $[c^{j-1}, c^j)$. By the hitting property discussed before, for any message bit, there is at least one q-tuple in the support of the decoder's queries which is still good with constant probability under the induced distribution. By (1) and (2) above, there must be at least $\Omega(md)$ good pairs with distance in $[d, cd]$, and this further implies that there exists a j such that P_j contains a constant fraction of good pairs. Now a packing argument implies that $n = O(\log m)$.

We remark that the random deletion channel (described above) that we use to establish the lower bound does not depend on anything about the codeword or the entire coding and decoding scheme. Thus, in contrast to the same bound for Hamming LDC, our lower bound continues to apply in private-key settings where the encoder and decoder share secret randomness.

Linear 2-query LDC The case of linear/affine codes is more involved. Here, we first use Fourier analysis to argue that if a pair of codeword bits is good for decoding a message bit, then the message bit must have non-trivial correlation with some parity of the codeword bits. However, since the code itself is linear or affine, this non-trivial correlation must be 1. By the hitting property, for any i'th message bit there exists a j_i such that a constant fraction of the pairs in P_{j_i} are good for i. By rearranging the message bits, without loss of generality we can assume that $j_1 \leq j_2 \leq \cdots \leq j_n$.

Now, for any i and P_{j_i} we have two cases: the message bit can have correlation 1 either with a single codeword bit, or with the parity of the two codeword bits. By averaging, at least one case consists of a constant fraction of the pairs in P_{j_i}. By another averaging, at least a constant fraction of the message bits fall into one of the above cases, so eventually we have two cases: (a) a constant fraction of the message bits each has correlation 1 with a constant fraction of all codeword bits, or (b) a constant fraction of the message bits each has correlation 1 with the parity of a constant fraction of the pairs in P_{j_i}.

The first case is easy since any codeword bit cannot simultaneously have correlation 1 with two different message bits, hence this implies we can only have a constant number of message bits. The second case is harder, where we use a delicate combinatorial argument to reduce to the first case. Specifically, for any such message bit i we can consider the bipartite graph G_i on $2m$ vertices induced by the good pairs in P_{j_i}, thus any such graph has bounded degree (since the distance of the pairs is bounded) and is dense in the sense that the edges take up a constant fraction of all possible edges.

For simplicity let us assume that having correlation 1 means that the two bits are the same as functions. Roughly, we use the dense property of these graphs to show the following: (c) there is an index $i = \Omega(n)$ and a right vertex $W \in G_i$ which is connected to a set T of $\Omega(c^{j_i})$ left vertices in G_i, and (d) there are $\Omega(n)$ indices $i' \leq i$ such that in each $G_{i'}$, the same set T is connected to a set $U_{i'}$ of $\Omega(c^{j_i})$ neighbors. By (c), all the codeword bits in T must be the same, and they are all contained in an interval of length c^{j_i}. Then by (d), all the codeword bits in $U_{i'}$ for different i' must be disjoint, since the parity of them with some bits in T equals a different message bit. Now notice that for any $i' \leq i$, all pairs in $P_{j'_i}$ have distance at most $c^{j'_i} \leq c^{j_i}$. This implies all the bits of all $U_{i'}$ are contained in an interval of length $2c^{j_i}$, which readily gives that $n = O(1)$.

The case of $q \geq 3$ We now generalize the above strategy to the case of $q \geq 3$. Consider the case of $q = 3$ for example. Now any query is a triple and we use (d_1, d_2) to stand for the distances of the two adjacent intervals in the query. If we can show similar properties as before, i.e., for any triple with distance (d_1, d_2) in the induced distribution: (1) The probability mass of any element is $O(\frac{1}{md_1d_2})$, and (2) With high probability, the corresponding triple will have distance (d'_1, d'_2) such that $d'_1 \in [d_1, cd_1], d'_2 \in [d_2, cd_2]$ for some constant $c = c(\delta, \varepsilon)$, then a similar argument would yield the bound of $n = O(\log^2 m)$, and for general q (at least constant q) the bound of $n = O(\log^{q-1} m)$.

However, unlike the case of $q = 2$, another tricky issue arises here. The issue is that with the error distribution discussed above, while we can ensure that for any *pair* of indices in the q-tuple, its marginal distribution behaves as before, the joint distribution of the q-tuple in the induced distribution behaves differently than what we expect. The reason is that (e.g., for $q = 3$) the two intervals with distance d_1 and d_2 are correlated under the error distribution. Specifically, the random deletion of each codeword bit again leads to a concentration phenomenon, thus conditioned on the number of deletions in the first interval, the parameter p is no longer uniformly distributed in the interval $[\frac{\delta}{8}, \frac{\delta}{4}]$, but rather pretty concentrated in a much smaller interval. This in turn affects the induced distribution of the second interval. Specifically, under this error distribution the bound on the probability in (1) becomes $O(\frac{\sqrt{d}}{md_1d_2})$, where $d = d_1 + d_2$. If we simply apply this bound, it will lead to (coincidentally or uncoincidentally) almost exactly the same bound as for Hamming LDC, thus we don't get any significant improvement.

To get around this and prove strong lower bounds for Insdel LDCs, we introduce additional random deletion processes to "break" the correlations discussed above. Towards this, we add another layer of deletions on top of the previous two operations: we first divide the codeword evenly into blocks of size s, and then for each block, we independently pick a parameter p uniformly from $[\frac{\delta}{8}, \frac{\delta}{4}]$ and delete each bit of this block independently with probability p. The idea is that, if for a 3-query it happens that one block is completely contained in one interval, then since the deletion process in that block is independent of the other blocks, the induced distribution of that interval is also more or less independent of the other interval.

However, this comes with another tricky issue: how to pick the size s. If s is too large, then for queries with small intervals, both intervals can be contained in the same block, and the deletion process would be exactly the same as before, which defeats the purpose of using blocks. On the other hand, if s is too small, then for queries with large intervals, the concentration and correlation phenomenon will happen again, which also defeats the purpose of using blocks. Since the intervals of the queries can have arbitrary distance, our solution is to actually use $O(\log m)$ layers of deletions, where for the j'th layer we use a block size of say 2^j. This ensures that for any query there is an appropriate block size, and in the analysis we can first condition on the fixing of all other layers, and argue about this layer.

Yet there is another price to pay here: since we are only allowed at most δm deletions, in each layer we cannot delete each bit with constant probability. Therefore for these layers we need to pick a parameter p uniformly from $[\frac{\delta}{8\log m}, \frac{\delta}{4\log m}]$. We call this deletion *type* **3**. This blows up our bound of the probability in (1) by a polylog factor, and we get a bound of $n = O(\log^{2q-3} m)$.

We note that in all the discussions so far, our error distributions do not depend on anything about the codeword or the entire coding and decoding scheme, thus all these results apply in settings where the encoder and decoder share secret randomness (private-key), which makes our lower bounds stronger. On the other hand, by exploiting the decoder's strategy, we can actually improve our bounds for the case of $q \geq 3$ (but the improved lower bounds no longer hold in the private-key setting). This time, we add another $O(\log m)$ layers of deletions on top of the previous three operations, where for the j'th layer we again use a block size of say 2^j. However, for these $O(\log m)$ layers the deletion parameter p is not picked from $[\frac{\delta}{8\log m}, \frac{\delta}{4\log m}]$, but rather uniformly from $[\frac{\delta p_j}{8}, \frac{\delta p_j}{4}]$, where p_j is the probability that the decoder uses a query whose first interval has distance in $[2^{j-1}, 2^j)$. We call this deletion *type* **4**. Notice that since $\sum_j p_j = 1$ the expected number of total deletions for this operation is still at most $\frac{\delta m}{4}$.

To get some intuition of why this helps us, consider the extreme case where all the queries used by the decoder have exactly the same distance for the first interval. Since there is no other distance for the first interval, we should not assign any probability mass of deletions to blocks of a different size, but should instead use the same block size, and delete each bit with probability p chosen uniformly from say $[\frac{\delta}{8}, \frac{\delta}{4}]$. This corresponds to the case where some $p_j = 1$, and the above strategy is a natural generalization. In the meantime, we still need all previous deletion types to take care of the other intervals. We show that under this deletion process we can replace one $\log m$ factor in the probability of (1) by $1/p_j$ (see Corollary 5.2.1 for a formal statement), and overall this leads to a bound of $n = O(\log^{2q-4} m)$ for $q \geq 3$.

5.1.2.2 LDCs with Randomized Encoding

In this section, we describe the ideas behind our constructions of LDCs with randomized encoding. We first explain how to can handle Hamming errors. Then we show how these constructions can be extended to edit errors.

Hamming Error We start with constructions for Hamming errors and let the fraction of errors be some constant δ. To take advantage of a randomized encoding, our approach is to let the encoder perform a *random permutation* of the codeword. Assuming the adversary does not know the randomness of the encoding, this effectively reduces adversarial errors into random errors, in the following sense: if we look at any subset of coordinates of the codeword before the random permutation, then the expected fraction of errors in these coordinates after the random permutation is also δ. A stronger statement also follows from concentration bounds that with high probability this fraction is not far from δ, and in particular is also a constant. This immediately suggests the following encoding strategy: first partition the message into blocks, then encode each block with a standard asymptotically good code, and finally use a random permutation to permute all the bits in all resulted codewords. Now to decode any target bit, one just needs to query the bits in the corresponding codeword for the block that contains this bit. As the error fraction is only a constant, a concentration bound for *random permutations* shows that in order to achieve success probability $1 - \varepsilon$, one needs block length $O(\log(1/\varepsilon))$ and this is also the number of queries needed. To ensure that the adversary does not learn any information about the random permutation by looking at the codeword, we also use the shared randomness to add a mask to the codeword (i.e., we compute the XOR of the actual codeword with a random string). This gives our codes for a fixed failure probability, in the model of shared randomness.

To modify our construction to the model of an oblivious channel, note that here we don't need a random mask anymore, but the encoder has to tell the decoder the random permutation used. However the description of the random permutation itself can be quite long, and this defeats our purpose of local decoding. Instead, we use a *pseudorandom permutation*, namely an almost $\kappa = \Theta(\log \frac{1}{\varepsilon})$ wise independent permutation with error $\varepsilon/3$. Such a permutation can be generated by a short random seed with length $r = O(\kappa \log n + \log(1/\varepsilon))$, and thus is good enough for our application. The encoder will first run the encoding algorithm described previously to get a string y with length $m/2$, and then concatenate y with an encoded version $z \in \{0,1\}^{m/2}$ of the random seed. The decoder will first recover the seed and then perform local decoding as before. To ensure the seed itself can be recovered by using only local queries, we encode the seed by using a concatenation code where the outer code is a (m_1, n_1, d_1) Reed-Solomon code with alphabet size $\mathsf{poly}(m)$, and the inner code is an (m_2, n_2, d_2) asymptotically good code for $O(\log m)$ bits, where $m_1 m_2 = m/2, n_1 n_2 = r$, $d_1 = m_1 - n_1 + 1$, $m_2 = O(\log m)$. That is, each symbol of the outer code is encoded into another block of $O(\log m)$ bits. Now to recover the seed, the decoder first randomly chooses $8n_1$ blocks of the concatenated codes and decodes the symbols of the outer code. These decoded symbols will form a new (shorter) Reed-Solomon code, because they are the evaluations of a degree n_1 polynomial on $8n_1$ elements in $\mathbb{F}_2^{n_2}$. Furthermore, this code is enough to recover the seed, because the seed is short and with high probability there are only a small constant fraction of errors in the decoded symbols. So the decoder can perform a decoding of the new Reed-Solomon code to recover the random seed with high probability. Note that the number of queries of this decoding is $8n_1 m_2 = O(r)$.

We now turn to our constructions for flexible failure probability. Here we want to achieve failure probability from a constant to say 2^{-n}. For each fixed failure probability we can use the previous construction, and this means the block size changes from a constant to $O(n)$. Instead of going through all of these sizes, we can just choose

$O(\log n)$ sizes and ensure that for any desired failure probability ε, there is a block size that is at most twice as large as the size we need. Specifically, the block sizes are $2^i, i = 1, 2, \ldots, \log n$. In this way, we have $O(\log n)$ different codewords, and we combine them together to get the final codeword. Now for any failure probability ε, the decoder can look for the corresponding codeword (i.e., the one where the block size is the smallest size larger than $1/\varepsilon$) and perform local decoding. However, we cannot simply concatenate these codewords together since otherwise δ fraction of errors can completely ruin some codeword. Instead, we put them up row by row into a matrix of $O(\log n)$ rows, and we encode each column of $O(\log n)$ bits with another asymptotically good binary code. Finally we concatenate all the resulted codewords together into our final codeword of length $O(n \log n)$. Note that now to recover a bit for some codeword, we need to query a whole block of $O(\log n)$ bits, thus the query complexity increases by a $\log n$ factor while the rate decreases by a $\log n$ factor.

Edit Error Our constructions for edit errors follow the same general strategy, but we need several modifications to deal with the loss of index information caused by insertions and deletions. Our construction for edit errors can achieve the same rate as those for Hamming case, but the query complexity for both models increases by a factor of polylog n. We now give more details. We start with the construction of an $(m = O(n), n, \delta = \Omega(1), q = \text{polylog}\, n \log(1/\varepsilon), \varepsilon)$ LDC for any $\varepsilon \in (0, 1)$, in the model with shared randomness. As in the case of Hamming errors, the shared randomness is used in two places: a random permutation π and some random masks to hide information. The construction has two layers.

For the first layer, view the message $x \in \{0,1\}^n$ as a sequence over the alphabet $\{0,1\}^{\log n}$ and divide it into $n/(n_0 \log n)$ small blocks each containing n_0 symbols from $\{0,1\}^{\log n}$. Then, we encode each block with Enc_0, which is an asymptotically good (m_0, n_0, d_0) code for Hamming errors over the alphabet $\{0,1\}^{\log n}$. Concatenating these $n/(n_0 \log n)$ codewords gives us a string of length $N = \frac{m_0 n}{n_0 \log n}$ over the alphabet $\{0,1\}^{\log n}$. We then permute these N symbols using π to get $y' = B_1 \circ B_2 \circ \cdots \circ B_N$ with $B_i \in \{0,1\}^{\log n}$. Since m_0/n_0 is a constant, we have $N < n$ for large enough n.

We are now ready to do the second layer of encoding. In the following, for each $i \in [N]$, $b_i \in \{0,1\}^{\log n}$ is the binary representation of i and $r_i \in \{0,1\}^{\log n}$ is a random mask shared between the encoder and decoder. $\mathbb{C}_0 : \{0,1\}^{2\log n} \to \{0,1\}^{10 \log n}$ is an asymptotically good code for edit errors and the $\oplus$ notation means bit-wise XOR. For each $i \in [N]$, we compute $B'_i = \mathbb{C}_0(b_i \circ (B_i \oplus r_i)) \in \{0,1\}^{10\log n}$. We output $y = B'_1 \circ B'_2 \circ \cdots \circ B'_N \in (\{0,1\}^{10\log n})^N$, which is of length $m = 10\frac{m_0}{n_0} n = O(n)$. Note that the use of the random masks r_i's is to hide the actual codeword, so that the adversary cannot learn any information about the permutation.

To decode a given message bit, we need to find the corresponding block. The natural idea to do this is by binary search. However, this may fail with high probability due to the constant fraction of edit errors. To solve this issue, we use the techniques developed by [97], which use a similar second layer encoding and give a searching algorithm with the following property: Even with δ-fraction of edit errors, at least $1 - O(\delta)$ fraction of the blocks can be recovered correctly with probability at least $1 - \mathsf{neg}(n)$. The algorithm makes a total of $\operatorname{polylog} n$ queries to the codeword for each search.

We now describe the decoding. Assume the bit we want to decode lies in the i-th block of x. Let $C_i \in (\{0,1\}^{\log n})^{m_0}$ be the codeword we get from encoding the i-th block using Enc_0. With the information of π, we can find out m_0 indices i_1 to i_{m_0} such that C_i is equal to $B_{\pi^{-1}(i_1)} \circ B_{\pi^{-1}(i_2)} \circ \cdots \circ B_{\pi^{-1}(i_{m_0})}$. The decoding algorithm calls the searching algorithm from [97] to find all blocks B'_{i_1} to $B'_{i_{m_0}}$ in the received codeword. We say a block is unrecoverable if the searching algorithm failed to find it correctly. By the same concentration bound used in the Hamming case and the result from [97], the fraction of unrecoverable blocks is bounded by a small constant with high probability. Thus, we can decode C_i correctly with the desired success probability. In this process, each search takes $\operatorname{polylog} n$ queries and $m_0 = O\Big(\log(1/\varepsilon)\Big)$ of searches are performed. The total number of queries made is thus $\operatorname{polylog} n \log(1/\varepsilon)$.

For the model of an oblivious channel, again we use a pseudorandom permutation π that can be generated by $O(\log m \log(1/\varepsilon))$ random bits. We use the same binary code as we used in the Hamming case to encode it and then view it as a string over the alphabet $\{0,1\}^{\log n}$. It is then concatenated with the code described previously before the second layer of encoding. After that, the same second layer of encoding is applied. The random masks used in the previous construction are no longer needed since the adversary can not see the codeword.

The construction for a flexible failure probability is also similar to the Hamming case. We write the codes before the second layer of encoding as a matrix M. The only difference is that, each element in the matrix M is now a symbol in $\{0,1\}^{\log n}$. We then encode the j-th column with an error correcting code over the alphabet $\{0,1\}^{\log n}$ to get a codeword z_j, and concatenate them to get z. After that, we do the second layer of encoding on z.

5.1.3 Preliminaries

Here we present some common notation and lemmas which we use throughout our proofs.

The indices i, j, k, ℓ are reserved for iterators; $c, \alpha, \beta, \gamma, \eta$ are reserved for constants; a, b, x, y, z are reserved for vectors or strings. For a string $y \in \{0,1\}^m$ and a subset $J \subseteq [m]$ of indices, we write $y_J \coloneqq \Big\{y_j : j \in J\Big\}$ for the restriction of y to J.

We may assume that decoder always queries exactly q indices. If some query uses a set of indices $Q' \subset [m]$ such that $|Q'| = q' < q$, we can replace Q' by $Q = Q' \cup \Big\{j_1, \dots, j_{q'-q}\Big\}$ where choices of $j_1, \dots, j_{q'-q} \in [m] \setminus Q'$ are arbitrary. In the actual decoding, the decoder will just ignore the extra symbols. Given a tuple $\Big\{k_0, \dots, k_{q-1}\Big\}$

with $k_0 < \cdots < k_{q-1}$, we also denote it by $\Big(k_0, d_1, d_2, \ldots, d_{q-1}\Big)$ where $d_i = k_i - k_{i-1}$ for $i = 1, 2, \ldots, q-1$. Note that this induces a bijection $\psi_{m,q}$ between $\mathcal{S}_{m,q} = \Big\{(k, d_1, \ldots, d_{q-1})\colon k, d_1, \ldots, d_{q-1} \geq 1, k + d_1 + \cdots + d_{q-1} \leq m\Big\}$ and $\binom{[m]}{q}$. Sometimes we will abuse the notation and write $Q \subseteq [m]^q$ while we actually mean the image of Q under $\psi_{m,q}$ (e.g. when we write $A \cap B$ where $A \subseteq \binom{[m]}{q}$ and $B \subseteq \mathcal{S}_{m,q}$), and vice versa.

Given a distribution $\mathcal{D}$ over some space Ω, denote by $\operatorname{supp}(\mathcal{D}) = \{\omega \in \Omega\colon \mathcal{D}(\omega) > 0\}$ the *support* of $\mathcal{D}$.

All logs are in base 2 unless otherwise specified. We write $\mathcal{H}(x) = -x\log x - (1-x)\log(1-x)$ for the binary entropy function, and we use the following upper bound (Proposition 1). The proof can be obtained via expanding $\mathcal{H}(x)$ into Taylor series around $x = 1/2$.

Proposition 1. *For $x \in (0, 1/2)$, we have $\mathcal{H}(1/2 + x) \leq 1 - (2(\ln 2)^2/3)x^2$.*

Basic Facts of Fourier Analysis. We start with a Boolean function from $\{0,1\}^n \to \{0,1\}$ and transform it to the $\{1,-1\}^n \to \{1,-1\}$ space by the transformation $u \mapsto (-1)^u$ for any bit u in the input or output.

Let f, g be two Boolean functions from Fourier space. We define their correlation to be $\mathsf{Corr}(f,g) = |\mathbb{E}_x f(x)g(x)| = |\Pr_x[f(x) = g(x)] - \Pr_x[f(x) \neq g(x)]|$. For a function f, its Fourier expansion is $\sum_{S\subseteq[n]} \hat{f}_S \chi_S(x)$, where $\chi_S(x) = \prod_{i\in S} x_i$ and $\hat{f}_S = \langle f, \chi_S\rangle = \mathbb{E}_x f(x)\chi_S(x)$. By this definition, for Boolean functions f, we always have $|\hat{f}_S| \leq 1$, since $f(u), \chi_S(u) \in \{-1,1\}$.

Proposition 2. *Let $f : \{-1,1\}^n \to \{-1,1\}$ and $C : \{-1,1\}^n \to \{-1,1\}^m$ be arbitrary functions. For every $Q \subseteq \binom{[m]}{q}$, if*

$$\sup_{S\subseteq Q} \left|\mathbb{E}_x f(x) \prod_{j\in S} y_j\right| < \frac{\varepsilon}{2^q},$$

where $y = C(x)$, then for any function $g : \{-1,1\}^q \to \{-1,1\}$, $\Pr_x\Big[g(y_Q) = f(x)\Big] < (1+\varepsilon)/2$.

Proof. We know that $g(y_Q) = \sum_{S \subseteq [q]} \hat{g}_S \chi_S(y_Q)$. So

$$\begin{aligned}
\left|\mathbb{E}_x g(y_Q) f(x)\right| &= \left|\mathbb{E}_x \sum_{S \subseteq [q]} \hat{g}_S \chi_S(y_Q) f(x)\right| \\
&= \left|\sum_{S \subseteq [q]} \mathbb{E}_x \hat{g}_S \chi_S(y_Q) f(x)\right| \\
&\leq \sum_{S \subseteq [q]} \left|\mathbb{E}_x \hat{g}_S \chi_S(y_Q) f(x)\right| \\
&\leq 2^q \sup_{S \subseteq [q]} \left|\mathbb{E}_x \hat{g}_S \chi_S(y_Q) f(x)\right| \\
&< 2^q \varepsilon / 2^q = \varepsilon.
\end{aligned}$$

So $\Pr_x \left[g(y_Q) = f(x)\right] < (1 + \varepsilon)/2$. □

Our analysis is based on designing a specific error pattern and deriving the necessary properties the decoder needs to have in order to perform well against such errors. In a high level, the error pattern is going to be in the following form. Given a codeword $y \in \{0,1\}^m$, we first obtain the *augmented codeword* $y' \in \{0,1\}^{2m}$ by appending m bits to the end of y. These bits may be random, and most often they will be independent and uniformly random bits. Then the augmented codeword undergoes a random deletion process, which we describe in details later in Section 5.2.1 and Section 5.2.2. For now, think of it as generating a subset $D \subseteq [2m]$ according to some distribution $\mathcal{D}$, and then deleting all bits from y' with indices in D. Finally, the string output by the deletion process is truncated at length m to obtain the final output $\widetilde{y}$. We will argue that with high probability, $\widetilde{y}$ has length exactly m (i.e. there are at most m deletions in total) and is close to the original codeword y (i.e. only a small number of deletions are introduced to the first half of y').

One would observe that we could equivalently augment the codeword to length m *after* the deletion process, and indeed this gives the same distribution (if the padded bits are i.i.d.). However, it turns out that our argument becomes cleaner if we view the deletions as if they also occur in the augmented part. Specifically, in the following definition we view the augmented bits as part of the codeword, as it is possible that in some situation they actually help the decoder to decode some message bits.

Definition 5.1.1. *For $i \in [n]$, define the set* Good_i *as*

$$\mathsf{Good}_i := \Big\{ Q \in \binom{[2m]}{q} : \exists \textit{a Boolean function } f\colon \{0,1\}^q \to \{0,1\} \\ \textit{such that } \Pr[f(C'(x)_Q) = x_i] \geq \frac{1}{2} + \frac{\varepsilon}{4} \Big\},$$

where the probability is over the uniform distribution of all messages and any possible randomness in the padded bits.

For $Q \in \binom{[2m]}{q}$, let $H_Q \subseteq [n]$ be a subset collecting all indices i for which Good_i contains Q. The following is a corollary to Theorem 2 in [46].

Proposition 3. $\forall Q \in \binom{[2m]}{q}, \left|H_Q\right| \leq q / \left(1 - \mathcal{H}(1/2 + \varepsilon/4)\right)$.

Proof. Let $I\left(\mathbf{x}_{H_Q}; C(\mathbf{x})_Q\right)$ denote the mutual information between $\mathbf{x}_{H_Q}$ and $C(\mathbf{x})_Q$. We have that

$$I\left(\mathbf{x}_{H_Q}; C(\mathbf{x})_Q\right) \leq \mathcal{H}\left(C(\mathbf{x})_Q\right) \leq q.$$

On the other hand,

$$\begin{aligned} I\left(\mathbf{x}_{H_Q}; C(\mathbf{x})_Q\right) &= \mathcal{H}\left(\mathbf{x}_{H_Q}\right) - \mathcal{H}\left(\mathbf{x}_{H_Q} \mid C(\mathbf{x})_Q\right) \\ &\geq \mathcal{H}\left(\mathbf{x}_{H_Q}\right) - \sum_{i \in H_Q} \mathcal{H}\left(x_i \mid C(\mathbf{x})_Q\right) \\ &\geq \left(1 - \mathcal{H}(1/2 + \varepsilon/4)\right) \cdot \left|H_Q\right|. \end{aligned}$$

Rearranging gives the result. □

A deletion pattern is a distribution $\mathcal{D}$ over subsets of $[2m]$. Let $D \subseteq [2m]$ be a set of deletions. We note that D induces a strictly increasing mapping $\phi_D\colon [2m - |D|] \to [2m]$, where $\phi_D(i) = \min\left\{ i' \in [2m] \colon \left|\overline{D} \cap [i']\right| \geq i \right\}$, or intuitively the index of i before the deletions are introduced.

Given $Q = \left\{k_0, \ldots, k_{q-1}\right\} \in \binom{[m]}{q}$, we denote $Q^D = \left\{\phi_D(k_0), \ldots, \phi_D(k_{q-1})\right\}$. Note that this is always well-defined when $|D| \leq m$. Most often we will work with a random $D \sim \mathcal{D}$ for some deletion pattern $\mathcal{D}$. In that case Q^D is a random variable, and sometimes we say that Q^D *corresponds to* Q under $\mathcal{D}$. If the event $Q^D \in \mathsf{Good}_i$ occurs, where Q is a random query of $\mathsf{Dec}(\cdot, m, i)$, we say that "$\mathsf{Dec}(\cdot, m, i)$ *hits* Good_i". In this thesis this event will be independent of the string given to Dec since Dec is non-adaptive, and $\mathcal{D}$ will be oblivious to the codeword.

Lemma 5.1.1. *Given a (q, δ, ε) insdel LDC, for any deletion pattern $\mathcal{D}$ such that $|D \cap [m]| \leq \delta m$ and $|D| \leq m$ for any $D \in \operatorname{supp}(\mathcal{D})$, and any $i \in [n]$, the probability that $\mathsf{Dec}(\cdot, m, i)$ hits Good_i is at least $3\varepsilon/2$.*

Proof. Consider a uniformly random message $x \in \{0,1\}^n$ and $y = C(x) \in \{0,1\}^m$. Let $y' \in \{0,1\}^{2m}$ be an augment of y, and denote by y^D the string obtained by deleting from y' all bits with indices in D and truncating at length m. Formally, $y^D_j = y'_{\phi_D(j)}$ for $j = 1, \ldots, m$. Note that this is well defined if $|D| \leq m$.

Denote by $\mathcal{E}$ the event "$\mathsf{Dec}\,(\cdot, m, i)$ hits Good_i". Conditioned on $\overline{\mathcal{E}}$, the decoder successfully outputs x_i with probability at most $1/2 + \varepsilon/4$, by definition of Good_i (even in the case where the decoder may output a random function).

When $|D \cap [m]| \leq \delta m$, we have that $\widetilde{\mathsf{ED}}(y, y^D) \leq \delta \cdot 2m$. By definition of a (q, δ, ε) insdel LDC, we have that

$$\begin{aligned}
\frac{1}{2} + \varepsilon &\leq \Pr\left[\mathsf{Dec}(y^D, m, i) = x_i\right] \\
&\leq \Pr\left[\mathsf{Dec}(y^D, m, i) = x_i \mid \mathcal{E}\right] \cdot \Pr\left[\mathcal{E}\right] + \Pr\left[\mathsf{Dec}(y^D, m, i) = x_i \mid \overline{\mathcal{E}}\right] \cdot \Pr\left[\overline{\mathcal{E}}\right] \\
&\leq \Pr\left[\mathcal{E}\right] + \left(\frac{1}{2} + \frac{\varepsilon}{4}\right) \cdot \left(1 - \Pr\left[\mathcal{E}\right]\right).
\end{aligned}$$

All probabilities above are over x, D, the randomness of the decoder and any possible randomness in the padded bits. Rearranging gives $\Pr\left[\mathcal{E}\right] \geq 3\varepsilon/(2 - \varepsilon) \geq 3\varepsilon/2$. $\square$

We will write $\mathbf{U}[a, b]$ for the uniform distribution over the interval $[a, b]$. For $n \in \mathbb{N}$

and $p \in [0,1]$, we will write $B(n,p)$ for the binomial distribution with n trials and success probability p. When p is a random variable with distribution $\mathcal{D}$, we will denote the resulting compound distribution by $B(n,\mathcal{D})$.

We use the following anti-concentration bound for the compound distribution $B(n,\mathbf{U}[a,b])$.

Lemma 5.1.2. *Let $n \in \mathbb{N}$, and $0 \le s < t \le 1$. Let X be a random variable following a compound distribution $B(n,\mathbf{U}[s,t])$. Then for any $0 \le k \le n$, we have*

$$\Pr\left[X = k\right] \le \frac{1}{(t-s)\,(n+1)}.$$

Proof. We can explicitly write the probability as

$$\Pr\left[X=k\right] = \frac{1}{t-s}\int_s^t \binom{n}{k} x^k\,(1-x)^{n-k}\,\mathrm{d}x \le \frac{1}{t-s}\int_0^t \binom{n}{k} x^k\,(1-x)^{n-k}\,\mathrm{d}x.$$

Denoting

$$I_k = \binom{n}{k}\int_0^t x^k(1-x)^{n-k}\,\mathrm{d}x,$$

we are going to show that $I_k \le 1/(n+1)$. Integration by parts gives

$$\begin{aligned} I_k &= \frac{1}{k+1}\binom{n}{k}\left(x^{k+1}(1-x)^{n-k}\Big|_0^t + (n-k)\int_0^t x^{k+1}(1-x)^{n-k-1}\,\mathrm{d}x \right) \\ &= \frac{1}{k+1}\binom{n}{k} t^{k+1}(1-t)^{n-k} + \frac{n-k}{k+1}\binom{n}{k}\int_0^t x^{k+1}(1-x)^{n-k-1}\,\mathrm{d}x \\ &= \frac{1}{n+1}\binom{n+1}{k+1} t^{k+1}(1-t)^{n-k} + \binom{n}{k+1}\int_0^t x^{k+1}(1-x)^{n-k-1}\,\mathrm{d}x \\ &= \frac{1}{n+1}\binom{n+1}{k+1} t^{k+1}(1-t)^{n-k} + I_{k+1}. \end{aligned}$$

Therefore by telescoping and the Binomial Theorem, we have

$$I_k = \sum_{j=k}^{n}\left(I_k - I_{k+1}\right) = \frac{1}{n+1}\sum_{j=k+1}^{n+1}\binom{n+1}{j} t^j (1-t)^{n+1-j} \le \frac{1}{n+1}.$$

□

Pseudorandom Permutation In our construction against oblivious channel, one key idea is to integrate a short description of a pseudorandom permutation into the codeword. We will utilize the construction of pseudorandom permutations from [126]. Here, we follow the definition from their work and introduce their results.

Definition 5.1.2 (Statistical Distance)**.** *Let D_1, D_2 be distributions over a finite set Ω. The variation distance between D_1 and D_2 is*

$$\|D_1 - D_2\| = \frac{1}{2}\sum_{\omega\in\Omega} |D_1(\omega) - D_2(\omega)|$$

We say that D_1 and D_2 are δ-close if $\|D_1 - D_2\| \leq \delta$.

Definition 5.1.3 (k-wise δ-dependent permutation)**.** *Let $\mathcal{F}$ be a family of permutations on n elements (allow repetition). Let $\delta > 0$, we say the family $\mathcal{F}$ is k-wise δ-dependent if for every k-tuple of distinct elements $\{x_1, x_2, \cdots, x_k\} \in [n]$, for $f \in \mathcal{F}$ chosen uniformly, the distribution $\{f(x_1), f(x_2), \cdots, f(x_k)\}$ is δ-close to uniform distribution*

In practice, we want to construction explicit families of permutations. Two related parameters are:

Definition 5.1.4 (Description length)**.** *The description length of a permutation family $\mathcal{F}$ is the number of random bits, used by the algorithm for sampling permutations uniformly at random from $\mathcal{F}$.*

Definition 5.1.5 (Time complexity)**.** *The time complexity of a permutation family $\mathcal{F}$ is the running time of the algorithm for evaluating permutation from $\mathcal{F}$*

It is known that we can construct families of k-wise almost independent permutations with short description length (optimal up to a constant factor).

Theorem 5.7 (Theorem 5.9 of [126])**.** *Let P_n denote the set of all permutation over $\{0,1\}^n$. There exists a k-wise δ-dependent family of permutation $\mathcal{F} \subset P_n$. $\mathcal{F}$ has description length $O(kn + \log\frac{1}{\delta})$ and time complexity* $\text{poly}(n, k, \log\frac{1}{\delta})$.

Concentration Bound In our proof, we use the following concentration bound from [127]

Lemma 5.1.3 ([127])**.** *Let $\pi : [n] \to [n]$ be a random permutation. For any set $S, W \subseteq [n]$, let $u = \frac{|W|}{n}|S|$. Then the following holds.*

- *for any constant $\delta \in (0, 1)$,*

$$\Pr[|\pi(S) \cap W| \leq (1-\delta)\mu] \leq e^{-\delta^2\mu/2},$$

$$\Pr[|\pi(S) \cap W| \geq (1+\delta)\mu] \leq e^{-\delta^2\mu/3}.$$

- *for any $d \geq 6\mu$, $\Pr[|\pi(S) \cap W| \geq d] \leq 2^{-d}$.*

5.2 Lower Bounds for Insdel LDCs

We now formally prove the lower bounds.

5.2.1 Bounds for 2-query Insdel LDCs

In this section, we prove lower bounds for 2-query insdel LDCs (Theorem 5.1 and Theorem 5.2).

We start by describing the error pattern. It is defined via the following random deletion process which is applied to the augmented codeword described in the last section i.e., we obtain the augmented codeword by appending m bits to the end of the codeword. Recall that after applying the random deletions below we can always truncate the final string back down to m bits.

Description of the error distribution

Step 1 Pick a real number $\beta \in [\frac{\delta}{8}, \frac{\delta}{4}]$ uniformly at random and then delete each bit $j \in [2m]$ independently with probability β.

Step 2 Pick an integer $e_2 \in \left\{0, 1, \ldots, \left\lfloor \frac{\delta m}{4} \right\rfloor\right\}$ uniformly at random and delete the first e_2 bits.

We remark that equivalently, the process can be thought of as maintaining a subset $D \subseteq [2m]$ of deletions and updating D in each step, and nothing is really deleted until the end of the process. We will sometimes take this view in later discussions. However, for readability we omitted the details as to how this set is updated.

The following proposition bounds the number of deletions introduced by the process.

Proposition 4. *Let $D \subseteq [2m]$ be a set of deletions generated by the process. Then we have*

- $\Pr\left[|D \cap [m]| > \delta m\right] \le 2^{-\Omega(m)}$,
- $\Pr\left[|D| > m\right] \le 2^{-\Omega(m)}$.

Proof. Let $D_1 \subseteq D$ be the subset of deletions introduced during Step 1. Since Step 2 introduces at most $\delta m/4$ deletions, it suffices to upper bound the probabilities of $|D_1 \cap [m]| > 3\delta m/4$ and $|D_1| > 3m/4$. Moreover, it suffices to prove the upper bounds for any fixed $\beta \in [\delta/8, \delta/4]$ picked in Step 1.

For the first item, notice that each bit $j \in [m]$ is deleted independently with probability $\beta \le \delta/4$. Thus by Hoeffding's inequality

$$\Pr\left[|D_1 \cap [m]| \ge \left(\frac{\delta}{4} + \frac{\delta}{2}\right) m\right] \le \exp\left(-\frac{\delta^2 m}{2}\right) = 2^{-\Omega(m)}.$$

The proof of the second item follows similarly from Hoeffding's inequality

$$\Pr\left[|D_1| \ge \frac{3m}{4}\right] \le \Pr\left[|D_1| \ge \left(\frac{\delta}{4} + \frac{1}{8}\right) \cdot 2m\right] \le \exp\left(-2\left(\frac{1}{8}\right)^2 \cdot 2m\right) = 2^{-\Omega(m)}.$$

□

In the following lemma, we fix an arbitrary query $\{k, \ell\} \in \binom{[m]}{2}$ of the decoder, with $k < \ell$, and represent it as (k, d) where $d = \ell - k$.

Let $(k', d') \in [2m] \times [2m]$ be the random pair that corresponds to (k, d) under the error distribution (see the discussion before Lemma 5.1.1). It should be clear that we always have $k' \geq k$ and $d' \geq d$. We prove some properties of the distribution of (k', d').

Lemma 5.2.1. *There exist two constants $c = c(\varepsilon) > 1$ and $c' = c'(\delta) > 0$ such that the following holds.*

- *The distribution of (k', d') is concentrated in the set $[2m] \times [d, cd]$ with probability $1 - \varepsilon$.*
- *Any support of (k', d') has probability at most $\frac{c'}{md}$.*

Proof. We prove the concentration result first. We will fix an arbitrary $\beta \in [\delta/8, \delta/4]$. By Hoeffding's inequality, we can take $c_0 = \sqrt{\ln(1/\varepsilon)/2}$ such that for any $n \in \mathbb{N}$ and $p \in [0, 1]$,

$$\Pr_{Y \sim B(n,p)} \left[Y \geq pn + c_0\sqrt{n} \right] \leq \varepsilon.$$

Take $c = 8c_0^2 = 4\ln(1/\varepsilon)$. Then $c > 1 + (c_0/(1-\beta))^2$ for any $\beta \leq \delta/4 < 1/2$. Let X denote the number of deletions occurred in $[d+1, cd]$, which follows a binomial distribution $B((c-1)d, \beta)$. Then by the choice of c_0 we have

$$\Pr\left[d' > cd\right] \leq \Pr\left[X \geq (c-1)d\right] \leq \Pr\left[X \geq \beta(c-1)d + c_0\sqrt{(c-1)d}\right] \leq \varepsilon.$$

Note that this holds for any choice of $\beta \leq \delta/4$, and thus the concentration result follows.

Now we turn to the anti-concentration result. Denote by $k' \mapsto k$ the event that the k'-th bit is retained and has index k after the deletion, and denote by $(k', d') \mapsto (k, d)$ the event $(k' \mapsto k) \wedge (k' + d' \mapsto k + d)$.

Write $\Pr_{S_1}[\cdot]$ for the error distribution after Step 1. Let X be the number deletions occurred in $\{k'+1, \dots, k'+d'-1\}$, which follows a compound distribution $B(d'-1, \mathbf{U}[\delta/8, \delta/4])$. We have

$$\begin{aligned}
\sum_{k''=0}^{k'} \Pr_{S_1}\left[\left(k', d'\right) \mapsto \left(k'', d\right)\right] &= \sum_{k''=0}^{k'} \Pr_{S_1}\left[k' \mapsto k''\right] \cdot \Pr_{S_1}\left[k'+d' \mapsto k''+d \;\middle|\; k' \mapsto k''\right] \\
&= \sum_{k''=0}^{k'} \Pr_{S_1}\left[k' \mapsto k''\right] \cdot \Pr_{S_1}\left[X = d'-d\right] \cdot \Pr_{S_1}\left[k'+d' \text{ is retained}\right] \\
&\le \Pr_{S_1}\left[k' \text{ is retained}\right] \cdot \frac{8}{\delta} \cdot \frac{1}{d'} \\
&\le \frac{8}{\delta} \cdot \frac{1}{d'}.
\end{aligned}$$

Here the first inequality is due to Lemma 5.1.2. Finally, averaging over e_2 gives

$$\begin{aligned}
\Pr\left[\left(k', d'\right) \mapsto (k, d)\right] &= \frac{8}{\delta m} \sum_{e_2=0}^{\delta m/8} \Pr_{S_1}\left[\left(k', d'\right) \mapsto (k+e_2, d)\right] \\
&\le \frac{8}{\delta m} \sum_{k''=0}^{k'} \Pr_{S_1}\left[\left(k', d'\right) \mapsto \left(k'', d\right)\right] \\
&\le \frac{8}{\delta m} \cdot \frac{8}{\delta} \cdot \frac{1}{d} \\
&= \frac{64}{\delta^2} \cdot \frac{1}{md}.
\end{aligned}$$

Therefore we can take $c' = 64/\delta^2$. $\square$

Before proceeding to prove the main theorems, we provide a dictionary of notations to facilitate the readers.

Notations. The sets S_i, S, T, U_i, V_i are subsets of $[m]$, where S_i, S, T, U_i are used to denote some set of the first indices (namely k for a pair $\{k, \ell\}$ with $k < \ell$), and V_i is used to denote some set of the second indices (namely ℓ for a pair $\{k, \ell\}$ with $k < \ell$). We have the following relation: $\forall i, U_i \subseteq T \subseteq S$.

The set Good_i is defined in Definition 5.1.1. The sets P_j, Q_i are subsets of $[2m] \times [2m]$, i.e., subsets of the pairs of indices that may or may not be used in the query. j is reserved for the index of some P_j.

The set $G_{k,i}$ is a subset of $[n]$, i.e., a subset of some indices of the message bits.

We recall the statement of our main theorem for 2-query linear insdel LDC.

Theorem 5.1. *For any $(2, \delta, \varepsilon)$ linear or affine Insdel LDC $C : \{0,1\}^n \to \{0,1\}^m$, we have $n = O_{\delta,\varepsilon}(1)$.*

To prove this theorem we first establish the following claim, which works for any $(2, \delta, \varepsilon)$ insdel (even non-linear/affine) LDC. Let c be the constant from Lemma 5.2.1. Consider all pairs of the form (k, d) in $[2m] \times [2m]$, and partition them into $t = \lceil \log_c(2m) \rceil = O(\log m)$ subsets $\{P_j\}$, where for any $j \in [t]$, $P_j = [2m] \times [c^{j-1}, c^j)$.

Claim 5.2.1. *There exists a constant $\gamma = \gamma(\delta, \varepsilon) \leq 1$ such that the following holds for any $(2, \delta, \varepsilon)$ insdel LDC. For any $i \in [n]$, there exists a $j \in [t]$ such that $|P_j \cap \mathsf{Good}_i| \geq \gamma m c^j$.*

Proof. Fix any $i \in [n]$. Let $D \subseteq [2m]$ be a random set of deletions generated by the random process. By Proposition 4, with probability $1 - 2^{-\Omega(m)} \geq 1 - \varepsilon/4$ for any large enough n (and thus also m), we have that $|D \cap [m]| \leq \delta m$ and $|D| \leq m$. Conditioned on this event, $\mathsf{Dec}(\cdot, m, i)$ hits Good_i with probability at least $3\varepsilon/2$ by Lemma 5.1.1. Therefore, unconditionally the probability that $\mathsf{Dec}(\cdot, m, i)$ hits Good_i is at least $3\varepsilon/2 - \varepsilon/4 = 5\varepsilon/4$

(if $|D| > m$ we simply assume that $\mathsf{Dec}(\cdot, m, i)$ never hits Good_i). This implies that for at least one (k, d) in the support of the queries of $\mathsf{Dec}(\cdot, m, i)$, the corresponding pair (k', d') hits Good_i with probability at least $5\varepsilon/4$.

Now by the first item of Lemma 5.2.1 and a union bound, $\mathsf{Dec}(\cdot, m, i)$ queries a pair in $\mathsf{Good}_i \cap ([2m] \times [d, cd])$ with probability at least $5\varepsilon/4 - \varepsilon = \varepsilon/4$. By the second item of Lemma 5.2.1, we must have

$$\left| \mathsf{Good}_i \cap ([2m] \times [d, cd]) \right| \geq \frac{\varepsilon m d}{4c'}.$$

Choose j' such that $c^{j'-1} \le d < c^{j'}$. Noticing that $[2m] \times [d, cd] \subseteq P_{j'} \cup P_{j'+1}$, for some $j \in \{j', j'+1\}$ we must have $\left|\mathsf{Good}_i \cap P_j\right| \ge \varepsilon md/(8c')$. Since $d \ge c^{j'-1} \ge c^{j-2}$, we can choose $\gamma = \varepsilon/(8c'c^2)$ and the claim follows. $\square$

By the definition of Good_i and Proposition 2, if a pair $\{k, \ell\} \in \mathsf{Good}_i$ ($k < \ell$), then one of the following cases must happen: (1) x_i has correlation at least $\varepsilon/8$ with y_k; (2) x_i has correlation at least $\varepsilon/8$ with y_ℓ; and (3) x_i has correlation at least $\varepsilon/8$ with $y_k \oplus y_\ell$. However, notice that the code is a linear or affine code, thus every bit in $C(x)$ is a linear or affine function of x, which has correlation either 1 or 0 with any x_i. Furthermore the inserted bits are independent, uniform random bits. Therefore in any of these cases, the correlation must be 1 and the bits involved must not contain any inserted bit.

Thus, for any $i \in [n]$ and the corresponding $j \in [t]$ guaranteed by Claim 5.2.1, by averaging we also have three cases: (1) P_j has at least $\gamma mc^j/4$ pairs such that the first bit has correlation 1 with x_i; (2) P_j has at least $\gamma mc^j/4$ pairs such that the second bit has correlation 1 with x_i; and (3) P_j has at least $\gamma mc^j/2$ pairs such that the parity of the pair of bits has correlation 1 with x_i.

By another averaging, we now have two cases: either (a) at least $n/4$ of the message bits fall into case (1) or (2) above, or (b) at least $n/2$ of the message bits fall into case (3) above. We prove Theorem 5.1 in each case.

Proof of Theorem 5.1 in case (a). In this case, without loss of generality assume that there is a subset $I \subseteq [n]$ with $|I| \ge n/4$ such that for any $i \in I$, the corresponding P_j has at least $\gamma mc^j/4$ pairs such that the first bit has correlation 1 with x_i. Notice that any bit in $C(x)$ can be the first bit for at most c^j pairs in P_j, this means that there must be at least $\gamma m/4$ different bits in $C(x)$ that has correlation 1 with x_i. Let this set be V_i and we have $|V_i| \ge \gamma m/4$.

Since for each $i \in I$ we have such a set V_i, and these sets must be disjoint (a bit cannot simultaneously have correlation 1 with x_i and $x_{i'}$ if $i \neq i'$), we have

$$\frac{n}{4} \cdot \frac{\gamma m}{4} \leq \sum_{i \in I} |V_i| = \left| \bigcup_{i \in I} V_i \right| \leq m.$$

This gives $n \leq 16/\gamma = O_{\delta,\varepsilon}(1)$. □

Proof of Theorem 5.1 in case (b). This is the harder part of the proof. Here, there is a subset $I \subseteq [n]$ with $|I| \geq n/2$ such that for any $i \in I$, the corresponding P_j has at least $\gamma m c^j/2$ pairs such that the parity of the pair of bits has correlation 1 with x_i. For each $i \in I$, let the set of these pairs be Q_i. Thus $|Q_i| \geq \gamma m c^{j_i}/2$, where for any $i \in I$, j_i is the corresponding index of P_j guaranteed by Claim 5.2.1. Let $|I| = n' \geq n/2$. By rearranging the message bits if necessary, without loss of generality we can assume that $I = [n']$ and $j_1 \leq j_2 \leq \cdots \leq j_{n'}$. Let S_i be the set of all first indices of Q_i which are connected to at least $\gamma c^{j_i}/4$ second indices. Formally, $S_i = \{k : |\{d : (k,d) \in Q_i\}| \geq \gamma c^{j_i}/4\}$.

Another way to view this is to consider the bipartite graph $G_i = ([m], [m], Q_i)$ (since the pairs in Q_i can only involve bits in $C(x)$). Then G_i has at least $\gamma m c^{j_i}/2$ edges, and the left and right degrees of G_i are both at most c^{j_i}. Now S_i is the subset of left vertices with degree at least $\gamma c^{j_i}/4$.

We have the following claim.

Claim 5.2.2. *For any $i \in [n']$, $|S_i| \geq \gamma m/4$.*

Proof. Since $|Q_i| \geq \gamma m c^{j_i}/2$, and each index in S_i is connected to at most c^{j_i} other indices, the claim follows by a Markov type argument. □

Now, for any index $k \in [m]$ and any index $i \in [n']$, we define the set $G_{k,i}$ to be the set of all indices $i' \leq i$ such that $k \in S_{i'}$. Formally, $G_{k,i} = \{i' \leq i : k \in S_{i'}\}$. We have the following claim:

Claim 5.2.3. *There exists a constant $\eta = \eta(\delta, \varepsilon) = \gamma/8$, an index $i \in [n']$ and a set $S \subseteq S_i$, such that*

- $|S| \geq \eta m$.
- *For any $k \in S$, we have $|G_{k,i}| \geq \eta n'$.*

Proof. First notice that $\sum_{k\in[m]} |G_{k,n'}| = \sum_{i\in[n']} |S_i|$. For a pair (i, k) with $i \in [n']$ and $k \in [m]$, we say it is good if $k \in S_i$ and $|G_{k,i}| \geq \gamma n'/8$. For any fixed $k \in [m]$, there are at least $|G_{k,n'}| - \gamma n'/8$ indices $i \in [n']$ such that (i, k) is good (this number may be negative, but that's still fine for us). To see this, let i^* be the smallest index such that $\left|G_{k,i^*}\right| = \gamma n'/8$ and notice that $|G_{k,n'}| - \gamma n'/8 = |G_{k,n'}| - |G_{k,i^*}|$ counts the number of i such that $i^* < i \leq n'$ and $k \in S_i$, i.e. the number of good pairs.

Therefore the total number of good pairs is at least

$$\sum_{k\in[m]} \left(|G_{k,n'}| - \frac{\gamma n'}{8}\right) = \sum_{i\in[n']} |S_i| - \frac{\gamma m n'}{8} \geq \frac{\gamma m n'}{8},$$

since for any $i \in [n']$, we have $|S_i| \geq \gamma m/4$.

By averaging, this implies that $\exists i \in [n']$, such that there are at least $\gamma m/8$ good pairs for this fixed i. Let S be the set of all good indices of k for this i, then we must have $|S| \geq \gamma m/8$ and for any $k \in S$, we have $k \in S_i$ and $|G_{k,i}| \geq \gamma n'/8$. Thus the claim holds. ☐

Now consider the index i and the set S guaranteed by the above claim. Recall j_i is the index j of P_j corresponding to i. We have the following claim.

Claim 5.2.4. *There exists a set $T \subseteq S$ and two indices $k_0, \ell_0 \in [m]$ such that the following holds.*

- $|T| \geq \frac{\eta\gamma c^{j_i}}{4}$.
- $T \subseteq [k_0, k_0 + c^{j_i}]$.

- *$\forall k \in T$, $C(x)_k \oplus C(x)_{\ell_0}$ has correlation 1 with x_i.*

Proof. Consider all pairs of indices $\{k, \ell\} \in Q_i$ $(k < \ell)$ with $k \in S$, and view it as a bipartite graph $G = (A, B, E)$ with indices k on the left, and indices ℓ on the right. Formally, $G = (A, B, E)$ with $A = \{a_1, \ldots, a_m\}$, $B = \{b_1, \ldots, b_m\}$ and edge $E = \{(a_k, b_\ell) : \{k, \ell\} \in Q_i, k < \ell\}$. Since for any $k \in S$, we have $k \in S_i$, we know that any a_k has degree at least $\gamma c^{j_i}/4$. Notice that there are m right vertices in B. Therefore there must exist an $\ell_0 \in [m]$ such that the node b_{ℓ_0} is connected to at least $\eta\gamma c^{j_i}/4$ vertices on the left, and we can let the set of all these vertices be $T = \{k : (a_k, b_{\ell_0}) \in E\}$. Since for any pair in Q_i, the parity of this pair of bits has correlation 1 with x_i, we have that $C(x)_k \oplus C(x)_{\ell_0}$ has correlation 1 with x_i for all $k \in T$.

Since the vertices in T are all connected to ℓ_0, and the distance $d = \ell - k$ for all pairs in Q_i is in $[c^{j_i - 1}, c^{j_i}]$, we must have that all indices $k \in T$ are in the range $[\ell_0 - c^{j_i}, \ell_0]$. Taking $k_0 = \ell_0 - c^{j_i}$ and the claim follows. $\square$

Now for any $i' \le i$, let $U_{i'} = S_{i'} \cap T$, and consider the set $V_{i'}$ of all indices $\ell \in [m]$ such that $\exists k \in U_{i'}$ with $\{k, \ell\} \in Q_{i'}$. In other words, $V_{i'}$ is set of neighbours of $U_{i'}$ in the bipartite graph $([m], [m], Q_{i'})$. We have the following claim.

Claim 5.2.5. *For any $i' \le i$, we have*

- *$V_{i'} \subseteq [k_0, k_0 + 2c^{j_i}]$.*
- *$|V_{i'}| \ge \gamma |U_{i'}|/4$.*

Proof. Since $U_{i'} \subseteq T$, and every pair of query in $Q_{i'}$ has distance at most $c^{j_{i'}} \le c^{j_i}$, we have $V_{i'} \subseteq [k_0, k_0 + 2c^{j_i}]$. Furthermore, since every index in $U_{i'}$ is connected to at least $\gamma c^{j_{i'}}/4$ indices in $V_{i'}$, while every index in $V_{i'}$ is connected to at most $c^{j_{i'}}$ indices in $U_{i'}$, we must have $|V_{i'}| \ge \gamma |U_{i'}|/4$. $\square$

Now, notice that for any $i' \leq i$, and any $\ell \in V_{i'}$, there exists some $k \in U_{i'} \subseteq T$ such that $C(x)_k \oplus C(x)_\ell$ has correlation 1 with $x_{i'}$. By Claim 5.2.4, $C(x)_k \oplus C(x)_{\ell_0}$ has correlation 1 with x_i. Thus $C(x)_\ell \oplus C(x)_{\ell_0}$ has correlation 1 with $x_i \oplus x_{i'}$, and $C(x)_\ell$ has correlation 1 with $x_i \oplus x_{i'} \oplus C(x)_{\ell_0}$. This means that for any two $i_1, i_2 \leq i$ with $i_1 \neq i_2$, we must have $V_{i_1} \cap V_{i_2} = \emptyset$. Therefore, all the $V_{i'}$'s for different i' must be disjoint. Thus we have the following inequality:

$$\frac{\gamma}{4}\left(\sum_{i' \leq i} |U_{i'}|\right) \leq \sum_{i' \leq i} |V_{i'}| \leq 2c^{j_i}.$$

Notice that $\sum_{k \in T} |G_{k,i}| = \sum_{i' \leq i} |S_{i'} \cap T| = \sum_{i' \leq i} |U_{i'}|$ and $\forall k \in T \subseteq S$, we have $|G_{k,i}| \geq \eta n'$. Thus

$$\sum_{i' \leq i} |U_{i'}| \geq \eta n' |T| \geq \frac{\eta^2 \gamma c^{j_i}}{4} n'.$$

Combining the two inequalities, we get $n' \leq 32/(\eta^2\gamma^2) = 2048/\gamma^4$. Since $n' \geq n/2$. This also implies that $n \leq 2n' = 4096/\gamma^4 = O_{\delta,\varepsilon}(1)$. $\square$

Next we prove a simple exponential lower bound for general 2-query insdel LDCs, i.e. Theorem 5.2. This should serve as a warm-up for the general $q \geq 3$ case.

Theorem 5.2. *For any $(2, \delta, \varepsilon)$ Insdel LDC $C : \{0,1\}^n \to \{0,1\}^m$, we have $m = \exp(\Omega_{\delta,\varepsilon}(n))$.*

Proof. Recall that $t = \lceil \log_c(2m) \rceil$ and $P_j = [2m] \times [c^{j-1}, c^j)$ for $j \in [t]$. For $j \in [t]$ and $i \in [n]$, we define $\beta_{j,i} = \frac{\left|P_j \cap \mathsf{Good}_i\right|}{\left|P_j\right|}$. Since $\left|P_j\right| = 2m(c^j - c^{j-1}) \leq 2mc^j$, by Claim 5.2.1 there is a constant $\gamma = \gamma(\delta, \varepsilon) < 1$ such that for any $i \in [n]$, there exists a $j \in [t]$ satisfying $\beta_{j,i} \geq \gamma$. By the Pigeonhole Principle, there exists a $j \in [t]$ such that $\beta_{j,i} \geq \gamma$ for at least n/t different i's. Fix this j to be j_0. We have

$$\sum_{i=1}^{n} \beta_{j_0,i} \geq \frac{\gamma n}{t}.$$

On the other hand, by Proposition 3 every pair (k,d) can belong to Good_i for at most $2/(1-\mathcal{H}(1/2+\varepsilon/4))$ different i's. Thus we have

$$\sum_{i=1}^{n} \left|P_{j_0} \cap \mathsf{Good}_i\right| \le \frac{2}{1-\mathcal{H}(1/2+\varepsilon/4)} \cdot \left|P_{j_0}\right| .$$

Altogether this yields

$$\frac{\gamma n}{t} \le \sum_{i=1}^{n} \beta_{j_0,i} = \sum_{i=1}^{n} \frac{\left|P_{j_0} \cap \mathsf{Good}_i\right|}{\left|P_{j_0}\right|} \le \frac{2}{1-\mathcal{H}(1/2+\varepsilon/4)}.$$

We have $n \le O_{\delta,\varepsilon}(t) = O_{\delta,\varepsilon}(\log m)$ and $m = \exp\Big(\Omega_{\delta,\varepsilon}(n)\Big)$. □

5.2.2 A More General Error Distribution

In this section we describe a general framework for designing error distributions, and instantiate it with two sets of parameters. The error distribution defined in this section will be used in the proof of Theorem 5.3. As before the error distribution is applied to the augmented codeword which is obtained by concatenating m bits to the end of the original codeword — the final codeword can be truncated back down to m bits after applying the random deletions below.

Given parameters $L \in \mathbb{N}$, $\mathbf{s} = (s_1, \ldots, s_L) \in [2m]^L$ and $\mathbf{h} = (h_1, \ldots, h_L) \in [0,1]^L$ such that

$$h \coloneqq \sum_{\ell=1}^{L} h_\ell \le \frac{1}{4},$$

we consider an error distribution $\mathcal{D}(L, \mathbf{s}, \mathbf{h})$ defined by the following process.

Description of the error distribution $\mathcal{D}(L, \mathbf{s}, \mathbf{h})$

Step 1 The first step introduces deletions through L layers. For the ℓ-th layer, we first divide $[2m]$ into $\lceil 2m/s_\ell \rceil$ consecutive blocks each of size s_ℓ, except for the last block which may have smaller size. For the b-th block in layer ℓ, we pick $q_{\ell,b} \in [0, h_\ell \delta]$ uniformly at random (independent of other blocks), and mark each bit in the block independently with probability $q_{\ell,b}$. Finally, we delete all bits which are marked at least once.

Step 2 Pick $\beta \in [0, \frac{1}{4}]$ uniformly at random and delete each bit independently with probability $\beta\delta$.

Step 3 Pick an integer $e_2 \in \left\{0, 1, \ldots, \left\lfloor \frac{\delta m}{4} \right\rfloor\right\}$ uniformly at random and delete the first e_2 bits.

By a union bound, after Step 1, each symbol is deleted with probability at most $h\delta$. We thus have the following proposition as an easy consequence of Hoeffding's inequality.

Proposition 5. *Let $D \subseteq [2m]$ be a set of deletions generated by $\mathcal{D}(L, \mathbf{s}, \mathbf{h})$. Then we have*

$$\Pr\left[|D \cap [m]| > \delta m\right] \leq \exp\left(-\frac{\delta^2 m}{8}\right), \text{ and } \Pr\left[|D| > m\right] \leq \exp\left(-(1-\delta)^2 m\right).$$

Proof. Let $D_2 \subseteq D$ be the subset of deletions introduced during Step 1 and Step 2. Since Step 3 introduces at most $\delta m/4$ deletions, it suffices to upper bound the probabilities of $|D_2 \cap [m]| > 3\delta m/4$ and $|D_2| > m$. Moreover, it suffices to prove the upper bounds after conditioned on an arbitrary set of deletion probabilities $q_{\ell,b} \in [0, h_\ell \delta]$ for each $\ell \in [L]$ and $b \leq \lceil 2m/s_\ell \rceil$, and $\beta \in [0, 1/4]$.

Under the conditional distribution, each bit $j \in [2m]$ is deleted with probability at most $(h + \beta)\,\delta \leq \delta/2$, and these deletions are independent of each other. The Hoeffding's inequality shows that

$$\Pr\left[|D_2 \cap [m]| > \left(\frac{\delta}{2} + \frac{\delta}{4}\right) m\right] \leq \exp\left(-2\left(\frac{\delta}{4}\right)^2 m\right) = \exp\left(-\frac{\delta^2 m}{8}\right),$$

$$\Pr\left[|D_2| > m\right] = \Pr\left[|D_2| > \left(\frac{\delta}{2} + \frac{1-\delta}{2}\right) \cdot 2m\right] \leq \exp\left(-(1-\delta)^2 m\right).$$

□

We fix an arbitrary query $Q = \left(k, d_1, \ldots, d_{q-1}\right)$ of the decoder, and let $\left(k', d'_1, \ldots, d'_{q-1}\right) \in [2m]^q$ be the random tuple that corresponds to Q under the error distribution $\mathcal{D}(L, \mathbf{s}, \mathbf{h})$ (see the discussion before Lemma 5.1.1). It should be clear that we always have $k' \geq k, d'_1 \geq d_1, \ldots, d'_{q-1} \geq d_{q-1}$.

Given the query Q, we can define for each $i \in [q-1]$ a subset $F_i \subseteq [L]$ of layers as

$$F_i = \left\{ \ell \in [L] \colon h_\ell \neq 0 \text{ and } \frac{d_i}{4} \leq s_\ell \leq \frac{d_i}{2} \right\}.$$

The following lemma is a generalization of Lemma 5.2.1.

Lemma 5.2.2. *Suppose that $F_i \neq \varnothing$ for each $i = 2, 3, \ldots, q-1$. The following propositions hold.*

- *Let $c = 4 \ln (q/\varepsilon)$. The distribution of $(k', d'_1, \ldots, d'_{q-1})$ is concentrated in the set $[2m] \times [d_1, cd_1] \times \cdots \times [d_{q-1}, cd_{q-1}]$ with probability $1 - \varepsilon$.*
- *For any $\left(\ell_2, \ldots, \ell_{q-1}\right) \in F_2 \times \cdots \times F_{q-1}$, any support of $\left(k', d'_1, \ldots, d'_{q-1}\right)$ has probability at most $\frac{(32/\delta)^q}{md_1} \prod_{i=2}^{q-1} \frac{1}{h_{\ell_i} d_i}$.*

Proof. For convenience, let $k'_0 = k'$ and $k'_i = k'_0 + \sum_{j=1}^{i} d'_i$. Similar to the proof of Lemma 5.2.1, we will write $k' \mapsto k$ for the event "the k'-th bit is not deleted and has index k after the deletion process", and write $\left(k', d'_1, \ldots, d'_{q-1}\right) \mapsto \left(k, d_1, \ldots, d_{q-1}\right)$ for the event $\bigwedge_{i=0}^{q-1} (k'_i \mapsto k_i)$.

To prove the first item, we are going to condition on an set of deletion probabilities (i.e. $q_{\ell,b}$ for each block and β), and e_2 in Step 3. For each $i \in [q-1]$, we consider a random variable X_i denoting the number of deletions introduced to $I_i \coloneqq \left\{k'_{i-1} + 1, \ldots, k'_i - 1\right\}$. It always holds that $0 \leq X_i \leq d'_i - 1$. Note that X_i does not depend on the deletions introduced in Step 3. Under the error distribution, each of these bits is deleted independently with probability at most $(h + \beta)\delta \leq \delta/2$. Thus, following an analysis similar to the proof of Lemma 5.2.1, the choice of $c = 4 \ln(q/\varepsilon)$

guarantees

$$\Pr[d'_i > cd_i] \leq \frac{\varepsilon}{q-1}.$$

Taking a union bound shows that

$$\Pr\left[(k', d'_1, d'_2, \ldots, d'_{q-1}) \in [2m] \times [d_1, cd_1] \times \cdots \times [d_{q-1}, cd_{q-1}]\right] \geq 1 - \varepsilon.$$

Recall that this holds for any set of deletion probabilities, and thus the first item follows.

We now show the second item: for any $\left(\ell_1, \ldots, \ell_{q-1}\right) \in F_1 \times \cdots \times F_{q-1}$, we have

$$\Pr\left[\left(k', d'_1, \ldots, d'_{q-1}\right) \mapsto \left(k, d_1, \ldots, d_{q-1}\right)\right] \leq \frac{(32/\delta)^q}{md_1} \cdot \prod_{i=2}^{q-1} \frac{1}{h_{\ell_i} d_i}.$$

Denote by $\Pr_{S_1,S_2}[\cdot]$ the error distribution before Step 3. Recall that for $i \in [q-1]$, X_i is the number of deletions introduced to the interval I_i, which is independent of Step 3. We first observe that Step 3 does not change the relative distances among the queried indices. Therefore we have

$$\begin{aligned}
&\Pr\left[\left(k', d'_1, \ldots, d'_{q-1}\right) \mapsto \left(k, d_1, \ldots, d_{q-1}\right)\right] \\
=&\frac{1}{\lfloor \delta m/4 \rfloor} \cdot \sum_{e_2=0}^{\lfloor \delta m/4 \rfloor} \Pr\left[\left(k', d'_1, \ldots, d'_{q-1}\right) \mapsto \left(k, d_1, \ldots, d_{q-1}\right) \;\middle|\; e_2\right] \\
=&\frac{1}{\lfloor \delta m/4 \rfloor} \cdot \sum_{e_2=0}^{\lfloor \delta m/4 \rfloor} \Pr_{S_1,S_2}\left[\left(k', d'_1, \ldots, d'_{q-1}\right) \mapsto \left(k + e_2, d_1, \ldots, d_{q-1}\right)\right] \\
\leq&\frac{8}{\delta m} \cdot \Pr\left[\left(X_1 = d'_1 - d_1\right) \wedge \cdots \wedge \left(X_{q-1} = d'_{q-1} - d_{q-1}\right)\right].
\end{aligned}$$

In the rest of the proof we will think of the error distribution as comprised of only Step 1 and 2. The chain rule of conditional probability gives

$$\begin{aligned}
&\Pr\left[\left(X_1 = d'_1 - d_1\right) \wedge \cdots \wedge \left(X_{q-1} = d'_{q-1} - d_{q-1}\right)\right] \\
=&\Pr\left[X_1 = d'_1 - d_1\right] \cdot \prod_{i=2}^{q-1} \Pr\left[X_i = d'_i - d_i \mid X_1 = d'_1 - d_1, \ldots, X_{i-1} = d'_{i-1} - d_{i-1}\right].
\end{aligned}$$

We finish the proof with 2 claims.

Claim 5.2.6. $\Pr[X_1 = d'_1 - d_1] \le 16/(\delta d'_1)$.

Proof of the claim. We are going to condition on the deletion probabilities $q_{\ell,b}$ and prove the same bound for any $q_{\ell,b} \in [0, h_\ell \delta]$. This clearly implies the claim. Moreover, under this conditional distribution, the deletions of individual bits in Step 1 are mutually independent.

Write $X_1 = X_1^{(1)} + X_1^{(2)}$ where $X_1^{(i)}$ $(i = 1, 2)$ is the number of deletions occurred in I_1, introduced in Step i. Since Step 1 deletes each bit independently with probability at most $h\delta \le \delta/4$, Hoeffding's inequality shows that

$$\Pr\left[X_1^{(1)} \ge \frac{1}{2} d'_1\right] \le \exp\left(-\frac{d'_1}{2}\right) \le \frac{1}{d'_1},$$

where the last inequality holds as long as $d'_1 \ge 1$. Also notice that given $X_1^{(1)}$, $X_1^{(2)}$ follows a compound distribution $B\left(d'_1 - X_1^{(1)} - 1, \mathbf{U}[0, \delta/4]\right)$. Therefore

$$\begin{aligned}
\Pr\left[X_1 = d'_1 - d_1\right] &= \mathbb{E}_{X_1^{(1)}}\left[\Pr\left[X_1^{(2)} = d'_1 - d_1 - X_1^{(1)}\right] \,\middle|\, X_1^{(1)}\right] \\
&\le \mathbb{E}_{X_1^{(1)}}\left[\frac{4}{\delta} \cdot \frac{1}{d'_1 - X_1^{(1)}}\right] \\
&\le \frac{4}{\delta} \cdot \frac{1}{d'_1/2} + \frac{4}{\delta} \cdot \Pr\left[X_1^{(1)} \ge \frac{1}{2} d'_1\right] \\
&\le \frac{16}{\delta} \cdot \frac{1}{d'_1}.
\end{aligned}$$

Here the first equality uses Lemma 5.1.2. □

Claim 5.2.7. $\forall 2 \le i \le q-1$, $\Pr\left[X_i = d'_i - d_i \mid X_1 = d'_1 - d_1, \ldots, X_{i-1} = d'_{i-1} - d_{i-1}\right] \le 32/(\delta h_{\ell_i} d_i)$.

Proof of the claim. For the i-th term where $2 \le i \le q - 1$, we recall that $\ell_i \in F_i$. Since all blocks in layer ℓ_i have size $s_{\ell_i} \le d_i/2 \le d'_i/2$ by the definition of F_i, there exists a block in layer ℓ_i which is completely contained in I_i. Suppose it is the b-th block and denote it by B_i. Note that we may also assume $|B_i| \ge d_i/4$ (if B_i is the last block and $|B_i| < d_i/4$, then the second last block is also contained in I_i and has size $s_{\ell_i} \geq d_i/4$).

Similar to the proof of the previous claim, we are going to condition on β and the deletion probabilities $q_{\ell,b'}$ for all $\ell \in [L]$ and $b' \leq \lceil 2m/s_\ell \rceil$, except for $q_{\ell_i,b}$ which is the deletion probability of B_i. Proving the same bound under the conditional distribution will imply the claim.

Write $X_i = X_{i,B} + X'_{i,B} + X_{i,\varnothing}$ where $X_{i,B}$ is the number of deletions introduced to B_i by layer ℓ_i, $X'_{i,B}$ is the number of deletions introduced to B_i by other sources, and $X_{i,\varnothing}$ is the number of deletions introduced to $I_i \setminus B_i$.

A crucial observation is that given $X'_{i,B}$, $X_{i,B}$ is independent of the X_j's for $j \neq i$, and follows a compound distribution $B\left(|B_i| - X'_{i,B}, \mathbf{U}[0, h_{\ell_i}\delta]\right)$. Similar to the analysis for $X_1^{(1)}$, since each bit is deleted independently with probability at most $(h+\beta)\delta \leq \delta/2$ during Step 1 and 2, Hoeffding's inequality implies

$$\Pr\left[X'_{i,B} \geq \frac{3}{4}|B_i|\right] \leq \exp\left(-\frac{|B_i|}{8}\right) \leq \frac{4}{|B_i|},$$

where the last inequality holds as long as $|B_i| \geq 1$. Therefore we have

$$\begin{aligned}
&\Pr\left[X_i = d'_i - d_i \mid X_1 = d'_1 - d_1, \ldots, X_{i-1} = d'_{i-1} - d_{i-1}\right] \\
=&\mathbb{E}_{X'_{i,B}, X_{i,\varnothing}}\left[\Pr\left[X_i = d'_i - d_i \mid X_1 = d'_1 - d_1, \ldots, X_{i-1} = d'_{i-1} - d_{i-1}\right] \,\middle|\, X'_{i,B}, X_{i,\varnothing}\right] \\
=&\mathbb{E}_{X'_{i,B}, X_{i,\varnothing}}\left[\Pr\left[X_{i,B} = d'_i - d_i - X'_{i,B} - X_{i,\varnothing}\right] \,\middle|\, X'_{i,B}, X_{i,\varnothing}\right] \\
\leq&\mathbb{E}_{X'_{i,B}, X_{i,\varnothing}}\left[\frac{1}{h_{\ell_i}\delta} \cdot \frac{1}{|B_i| - X'_{i,B} + 1}\right] \\
\leq&\frac{1}{h_{\ell_i}\delta} \cdot \left(\frac{1}{|B_i|/4} + \frac{4}{|B_i|}\right) = \frac{8}{h_{\ell_i}\delta} \cdot \frac{1}{|B_i|} \leq \frac{32}{\delta} \cdot \frac{1}{h_{\ell_i} d_i}.
\end{aligned}$$

Here the first inequality is again due to Lemma 5.1.2. □

Putting everything together, we have shown that

$$\begin{aligned}\Pr\left[\left(k', d'_1, \ldots, d'_{q-1}\right) \mapsto \left(k, d_1, \ldots, d_{q-1}\right)\right] &\leq \frac{8}{\delta m} \cdot \left(\frac{16}{\delta} \cdot \frac{1}{d'_1}\right) \cdot \prod_{i=2}^{q-1} \left(\frac{32}{\delta} \cdot \frac{1}{h_{\ell_i} d_i}\right) \\ &\leq \frac{(32/\delta)^q}{m d_1} \cdot \prod_{i=2}^{q-1} \frac{1}{h_{\ell_i} d_i}.\end{aligned}$$

□

In the rest of the section, we instantiate $\mathcal{D}(L, \mathbf{s}, \mathbf{h})$ with two specific sets of parameters, which we now describe.

An adversarial error distribution for $q \geq 3$ We now define a non-oblivious error distribution $\mathcal{D}_{adv,i}$ which may depend on the decoder Dec. Analyzing $\mathcal{D}_{adv,i}$ allows us to derive tighter lower bounds on the codeword length m for a Insdel LDC with query complexity q. However, because the distribution is not oblivious the stronger lower bounds derived from $\mathcal{D}_{adv,i}$ no longer apply in the private-key setting.

Fix $i \in [n]$. Let $\left(K, D_1, D_2, \ldots, D_{q-1}\right)$ be the random variable that corresponds to queries of $\mathsf{Dec}(\cdot, m, i)$. For $1 \leq \tau \leq \lceil \log(2m) \rceil$, let $p_{\tau,i}$ be the probability that $2^{\tau-1} \leq D_2 < 2^{\tau}$. Thus, $p_{\tau,i}$ is the probability that the decoder for the i-th bit ($\mathsf{Dec}(\cdot, m, i)$) queries a tuple $(k, d_1, \ldots, d_{q-1})$ such that $2^{\tau-1} \leq d_2 < 2^{\tau}$. We have $\sum_{\tau=1}^{\lceil \log(2m) \rceil} p_{\tau,i} = 1$.

We take $L = 2L_0$ where $L_0 = \lceil \log(2m) \rceil$. The vectors $\mathbf{s} = (s_1, \ldots, s_L)$ and $\mathbf{h} = (h_1, \ldots, h_L)$ are defined as follows.

- $\forall\, 1 \leq \ell \leq L_0$, $s_\ell = 2^\ell$, and $h_\ell = 1/(8L_0)$.
- $\forall\, 1 \leq \tau \leq L_0$, $s_{L_0+\tau} = 2^{\tau-2}$, and $h_{d+L_0} = p_{\tau,i}/8$.

We define the adversary error distribution depending on $\mathsf{Dec}(\cdot, m, i)$ as $\mathcal{D}_{adv,i} := \mathcal{D}(L, \mathbf{s}, \mathbf{h})$. For this error distribution we also have

$$h = \sum_{\ell=1}^{L_0} h_\ell + \sum_{\tau=1}^{L_0} h_{\tau+L_0} = L_0 \cdot \frac{1}{8L_0} + \frac{1}{8} \cdot \sum_{\tau=1}^{L_0} p_{\tau,i} = \frac{1}{4}.$$

Let $\left(k, d_1, d_2, \ldots, d_{q-1}\right)$ be an arbitrary query in the support of $\mathsf{Dec}(\cdot, m, i)$ and $1 \le \tau_0 \le t$ be the integer such that $2^{\tau_0 - 1} \le d_2 < 2^{\tau_0}$. We set $\ell_2 = L_0 + \tau_0$. Since $s_{\ell_2} = 2^{\tau_0 - 2}$, we have $d_2/4 \le s_{\ell_2} \le d_2/2$. Thus, $\ell_2 \in F_2$ with $h_{\ell_2} = p_{\tau_0,i}/8$.

For $3 \le j \le q-1$, we set $\ell_j = \lceil \log d_j \rceil - 2 \in F_j$.Thus, $h_{\ell_j} = 1/(8L_0) \ge 1/(8(\log m + 2))$ for $3 \le j \le q-1$. We have the following corollary to Lemma 5.2.2.

Corollary 5.2.1. *Let $\left(k', d'_1, \ldots, d'_{q-1}\right)$ be the random tuple that corresponds to the query $\left(k, d_1, \ldots, d_{q-1}\right)$ under error distribution $\mathcal{D}_{adv,i}$. Let τ_0 be the integer such that $2^{\tau_0-1} \le d_2 < 2^{\tau_0}$. Then any support of $\left(k', d'_1, \ldots, d'_{q-1}\right)$ has probability at most*

$$\frac{(32/\delta)^q}{m} \cdot \frac{8^{q-2}(\log m + 2)^{q-3}}{p_{\tau_0,i}} \cdot \prod_{\ell=1}^{q-1} \frac{1}{d_\ell}.$$

5.2.3 Lower Bounds For Insdel LDCs with $q \ge 3$

In this section, we formally prove Theorem 5.3. We assume the error distribution is $\mathcal{D}_{adv,i}$ introduced in section 5.2.2. Following the notation from section 5.2.2, let $p_{\tau,i}$ be the probability that $\mathsf{Dec}(\cdot, m, i)$ queries a tuple $(k, d_1, \ldots, d_{q-1})$ such that $2^{\tau-1} \le d_2 < 2^{\tau}$. We have $\sum_{\tau=1}^{\lceil \log(2m) \rceil} p_{\tau,i} = 1$. Take $\eta = (256/\delta)^q$, $c = 4\ln(q/\varepsilon) \ge 2$ and denote $t = \lceil \log_c(2m) \rceil$. For $j_1, j_2, \ldots, j_{q-1} \in [t]$, denote

$$P_{j_1,\ldots,j_{q-1}} = [2m] \times [c^{j_1-1}, c^{j_1}) \times \cdots \times [c^{j_{q-1}-1}, c^{j_{q-1}}).$$

Let $I_\tau = \{P_{j_1,\ldots,j_{q-1}} \colon 2^{\tau-1} \le c^{j_2} \le c^2 2^\tau\}$ be a set of subcubes. We define

$$\beta_{\tau,i} = \max_{P_{\mathbf{J}} \in I_\tau} \frac{|P_{\mathbf{J}} \cap \mathsf{Good}_i|}{|P_{\mathbf{J}}|}.$$

Thus, $\beta_{\tau,i}$ is the maximum fraction of good points in any subcube $P_{\mathbf{J}}$ in the set I_τ.

Claim 5.2.8. *For any $i \in [n]$, we have*

$$\sum_{\tau=1}^{t} \beta_{\tau,i} \ge \frac{\varepsilon}{8\eta(2c^2)^{q-1}(\log m + 2)^{q-3}}.$$

Proof. We fix $i \in [n]$. Let $Q = (k, d_1, \ldots, d_{q-1})$ be an arbitrary query in the support of $\mathsf{Dec}(\cdot, m, i)$, and let $Q' = (k', d'_1, \ldots, d'_{q-1})$ be the random tuple corresponding to $(k, d_1, \ldots, d_{q-1})$ under error distribution $\mathcal{D}_{adv,i}$.

For $1 \leq \ell \leq q-1$, we let j'_ℓ be the integer such that $c^{j'_\ell - 1} \leq d_\ell < c^{j'_\ell}$. We have $[d_\ell, cd_\ell] \subseteq [c^{j'_\ell - 1}, c^{j'_\ell}) \cup [c^{j'_\ell}, c^{j'_\ell + 1})$. Let U_Q be a set of 2^{q-1} tuples $\left(j_1, \ldots, j_{q-1}\right)$ such that $j_\ell \in \{j'_\ell, j'_\ell + 1\}$ for all $\ell \in [q-1]$ (if $j'_\ell = t$, fix $j_\ell = j'_\ell$). By Lemma 5.2.2, with probability at least $1 - \varepsilon$, we have

$$(k', d'_1, \ldots, d'_{q-1}) \in \bigcup_{\mathbf{J} \in U} P_{\mathbf{J}}.$$

Denote this event by $\mathcal{E}$. We now give an upper bound of the probability that Q' hits Good_i in terms of the $\beta_{\tau,i}$'s. Let $1 \leq \tau_Q \leq \lceil \log(2m) \rceil$ be the integer such that $2^{\tau_Q - 1} \leq d_2 < 2^{\tau_Q}$. Notice that $2^{\tau_Q - 1} \leq d_2 < c^{j'_2} < c^{j'_2 + 1}$ and $c^{j'_2 + 1} \leq c^2 d_2 < c^2 2^{\tau_Q}$. We have $2^{\tau_Q - 1} \leq c^{j'_2} < c^{j'_1 + 1} \leq c^2 2^{\tau_Q}$. Thus for any $\mathbf{J} \in U_Q$, we have $P_{\mathbf{J}} \in I_{\tau_Q}$. By our definition of $\beta_{\tau_Q,i}$, we have

$$\beta_{\tau_Q,i} \geq \frac{|P_{\mathbf{J}} \cap \mathsf{Good}_i|}{|P_{\mathbf{J}}|}.$$

By Corollary 5.2.1, any support of $(k', d'_1, \cdots, d'_{q-1})$ has probability at most

$$\frac{\eta (\log m + 2)^{q-3}}{m d_1 \ldots d_{q-1} p_{\tau_Q,i}}$$

for $\eta = (256/\delta)^q$.

The size of any subcube $P_{j_1,\ldots,j_{q-1}}$ is bounded by $2mc^{j_1} \cdots c^{j_{q-1}}$. Since for $\mathbf{J} \in U_Q$ we have $c^{j_\ell} \leq c^{j'_\ell + 1} \leq c^2 d_\ell$ for any $1 \leq \ell \leq q-1$, we have $|P_{\mathbf{J}}| \leq (c^2)^{q-1} \cdot 2md_1 \cdots d_{q-1}$.

Thus the probability that $(k', d'_1, \ldots, d'_{q-1})$ hits Good_i can be bounded by

$$\begin{aligned}
&\Pr\left[(k', d'_1, \ldots, d'_{q-1}) \text{ hits } \mathsf{Good}_i\right] \\
\leq &\Pr\left[\overline{\mathcal{E}}\right] + \Pr\left[(k', d'_1, \ldots, d'_{q-1}) \in \mathsf{Good}_i \cap \bigcup_{\mathbf{J} \in U_Q} P_{\mathbf{J}}\right] \\
\leq &\varepsilon + \frac{\eta(\log m + 2)^{q-3}}{m d_1 \cdots d_{q-1} p_{\tau_Q, i}} \cdot \beta_{\tau_Q, i} \cdot \sum_{\mathbf{J} \in U_Q} |P_{\mathbf{J}}| \\
\leq &\varepsilon + 2\beta_{\tau_Q, i} \cdot \frac{(2c^2)^{q-1}\eta(\log m + 2)^{q-3}}{p_{\tau_Q, i}}.
\end{aligned}$$

Denote by $\mu_i(\cdot)$ the probability distribution of the queries of $\mathsf{Dec}(\cdot, m, i)$. For the probability that $\mathsf{Dec}(\cdot, m, i)$ hits Good_i, we have

$$\begin{aligned}
&\Pr\left[\mathsf{Dec}(\cdot, m, i) \text{ hits } \mathsf{Good}_i\right] \\
= &\sum_{Q \in \binom{[2m]}{q}} \mu_i(Q) \cdot \Pr\left[\mathsf{Dec}(\cdot, m, i) \text{ hits } \mathsf{Good}_i \mid \mathsf{Dec}(\cdot, m, i) \text{ queries } Q\right] \\
\leq &\sum_{Q \in \binom{[2m]}{q}} \mu_i(Q) \cdot \left(\varepsilon + 2\beta_{\tau_Q, i} \cdot \frac{(2c^2)^{q-1}\eta(\log m + 2)^{q-3}}{p_{\tau_Q, i}}\right) \\
= &\sum_{Q \in \binom{[2m]}{q}} \mu_i(Q)\varepsilon + \sum_{Q \in \binom{[2m]}{q}} \mu_i(Q) \cdot 2\beta_{\tau_Q, i} \cdot \frac{(2c^2)^{q-1}\eta(\log m + 2)^{q-3}}{p_{\tau_Q, i}} \\
= &\varepsilon + (2c^2)^{q-1}\eta(\log m + 2)^{q-3} \cdot \sum_{\tau=1}^{\lceil \log(2m) \rceil} 2\beta_{\tau, i} \cdot \frac{1}{p_{\tau, i}} \cdot \sum_{Q\colon \tau_Q = \tau} \mu_i(Q) \\
\leq &\varepsilon + 2(2c^2)^{q-1}\eta(\log m + 2)^{q-3} \sum_{\tau=1}^{\lceil \log(2m) \rceil} \beta_{\tau, i}.
\end{aligned}$$

The last equality is due to the fact that for any $1 \leq \tau \leq \lceil \log(2m) \rceil$,

$$p_{\tau, i} = \sum_{Q\colon \tau_Q = \tau} \mu_i(Q).$$

By Proposition 5 and Lemma 5.1.1, for large enough n (and thus m) the probability that $\mathsf{Dec}(\cdot, m, i)$ hits Good_i is at least $3\varepsilon/2 - \varepsilon/4 = 5\varepsilon/4$. Thus

$$2(2c^2)^{q-1}\eta(\log m + 2)^{q-3} \sum_{\tau=1}^{\lceil \log(2m) \rceil} \beta_{\tau, i} \geq \varepsilon/4.$$

□

Claim 5.2.9. $\sum_{i=1}^{n}\sum_{\tau=1}^{\lceil \log(2m)\rceil} \beta_{\tau,i} \leq 3q\log c \cdot t^{q-1}/\left(1-\mathcal{H}(1/2+\varepsilon/4)\right).$

Proof. By Proposition 3, for any tuple $Q \in \binom{[m]}{k}$, Q is in Good_i for at most $\frac{q}{1-\mathcal{H}(1/2+\varepsilon/4)}$ different i's. Thus for any subcube $P_{\mathbf{J}}$, we have

$$\sum_{i=1}^{n}|P_{\mathbf{J}} \cap \mathsf{Good}_i| \leq \frac{q}{1-\mathcal{H}(1/2+\varepsilon/4)}|P_{\mathbf{J}}|\,.$$

Meanwhile, by the definition of $\beta_{\tau,i}$, we have

$$\beta_{\tau,i} = \max_{P_{\mathbf{J}}\in I_\tau}\frac{|P_{\mathbf{J}} \cap \mathsf{Good}_i|}{|P_{\mathbf{J}}|} \leq \sum_{P_{\mathbf{J}}\in I_\tau}\frac{|P_{\mathbf{J}} \cap \mathsf{Good}_i|}{|P_{\mathbf{J}}|}.$$

Combining the above two inequalities, we have

$$\sum_{i=1}^{n}\beta_{\tau,i} \leq \sum_{i=1}^{n}\sum_{P_{\mathbf{J}}\in I_\tau}\frac{|P_{\mathbf{J}} \cap \mathsf{Good}_i|}{|P_{\mathbf{J}}|} \leq \frac{q}{1-\mathcal{H}(1/2+\varepsilon/4)}|I_\tau|\,.$$

By the definition of I_τ, each subcube $P_{\mathbf{J}}$ belongs to at most $\lceil \log 2c^2\rceil \leq 3\log c$ consecutive I_τ's. Notice that the total number of subcubes is bounded by t^{q-1}. By counting the number of subcubes, we have

$$\sum_{\tau=1}^{\lceil \log(2m)\rceil}|I_\tau| \leq 3\log c\cdot t^{q-1}.$$

Thus,

$$\sum_{i=1}^{n}\sum_{\tau=1}^{\lceil \log(2m)\rceil}\beta_{\tau,i} \leq \frac{q}{1-\mathcal{H}(1/2+\varepsilon/4)}\cdot\sum_{\tau=1}^{t}|I_\tau| \leq \frac{q\cdot 3\log c\cdot t^{q-1}}{1-\mathcal{H}(1/2+\varepsilon/4)}.$$

□

Now we are ready to prove Theorem 5.3.

Theorem 5.3. *For any non-adaptive (q,δ,ε) Insdel LDC $C\colon \{0,1\}^n \to \{0,1\}^m$ with $q \geq 3$, we have the following bounds.*

$$m = \begin{cases}\exp(\Omega_{\delta,\varepsilon}(\sqrt{n})) \text{ for } q=3\text{; and}\\ \exp\left(\Omega\left(\frac{\delta}{\ln^2(q/\varepsilon)}\cdot\left(\varepsilon^3 n\right)^{1/(2q-4)}\right)\right) \text{ for } q\geq 4.\end{cases}$$

Proof. By Claim 5.2.8, we have

$$\sum_{i=1}^{n} \sum_{\tau=1}^{\lceil \log(2m) \rceil} \beta_{\tau,i} \geq \frac{n\varepsilon}{8\eta(2c^2)^{q-1}(\log m + 2)^{q-3}}.$$

Combined with Claim 5.2.9, we have

$$\frac{n\varepsilon}{8\eta(2c^2)^{q-1}(\log m + 2)^{q-3}} \leq \frac{q \cdot 3\log c \cdot t^{q-1}}{1 - \mathcal{H}(1/2 + \varepsilon/4)}.$$

Plugging in $\eta = (256/\delta)^q$ and $c = 4\ln(q/\varepsilon)$, and noticing that $t \leq \log m + 2$, $q \leq 2^q$, $3\log c \leq c^2$, for some large enough constant C we have

$$n \leq \frac{1}{\varepsilon(1 - \mathcal{H}(1/2 + \varepsilon/4))} \cdot \left(\frac{C\ln^2(q/\varepsilon)}{\delta}\right)^q (\log m + 2)^{2q-4}.$$

By Proposition 1, we have $1 - \mathcal{H}(1/2 + \varepsilon/4) = \Omega(\varepsilon^2)$. We can rewrite the above inequality as

$$m = \exp\left(\Omega\left(\left(\frac{\delta}{\ln^2(q/\varepsilon)}\right)^{\frac{q}{2q-4}} \cdot \left(\varepsilon^3 n\right)^{\frac{1}{2q-4}}\right)\right).$$

Thus, for $q = 3$, we have $m = \exp(\Omega_{\delta,\varepsilon}(\sqrt{n}))$. For $q \geq 4$, we have $\frac{q}{2q-4} \leq 1$ and $\left(\frac{\delta}{\ln^2(q/\varepsilon)}\right)^{\frac{q}{2q-4}} = \Omega\left(\frac{\delta}{\ln^2(q/\varepsilon)}\right)$. We can write

$$m = \exp\left(\Omega\left(\frac{\delta}{\ln^2(q/\varepsilon)} \cdot \left(\varepsilon^3 n\right)^{\frac{1}{2q-4}}\right)\right).$$

□

5.3 Constructions of LDCs with Randomized Encoding

In this section, we formally present our constructions of LDCs with randomized encoding. In this section, we follow the notations from [100], which first presented these proofs. We use k to denote the message length and n to denote the codeword length.

In our constructions, we will utilize the following greedily constructed code from [75].

Lemma 5.3.1 (Lemma 2 of [75]). *For small enough δ, there exists code $\mathbb{C} : \{0,1\}^n \to \{0,1\}^{4n}$ such that the edit distance between any two codewords is at least δ and for any codeword, every interval has at least half of 1's.*

Before the main construction, we also present a binary "locally decodable code" which can recover the whole message locally when the message length is sufficiently small.

Lemma 5.3.2. *For every sufficiently small constant $\delta > 0$ and every $k \leq cn$ with sufficiently small constant $c = c(\delta) < 1$, there is an explicit binary code that encodes a k bits message to an n bits codeword. The code has a randomized decoding algorithm such that if the received codeword has at most δn Hamming errors, then the decoder can compute the message with success probability $1 - 2^{-\Theta(k/\log n)}$ by only querying at most $q = O(k)$ bits of the received codeword.*

Proof. Consider the concatenation of an (n_1, k_1, d_1) Reed-Solomon code with alphabet $\{0,1\}^{n_2 = O(\log n_1)}$, $n_1 = n/n_2$, $k_1 = k/k_2$, and an explicit $(n_2, k_2 = n_2 - 2d_2(\log \frac{n_2}{d_2}), d_2 = \Theta(n))$ binary ECC.

The concatenated code is an $(n_1 n_2, k_1 k_2, d_1 d_2)$ code. Note that $n_1 n_2 = n$, $k_1 k_2 = k$. Also $d_1 = n_1 - k_1 + 1 = \Theta(n_1)$, $d_2 = O(n_2)$ so $d_1 d_2 = O(n)$.

We consider the following decoding algorithm. Given a codeword, we call the encoded symbols (encoded by the second code) of the first code as blocks. The algorithm randomly picks $8k_1$ blocks and query them. For each block, it calls the decoding function of the second code. After this we get $8k_1$ symbols of the first code. Then we use the decoding algorithm of a $(8k_1, k_1, 7k_1 + 1)$ Reed Solomon code to get the message.

Next we argue that this algorithm successes with high probability. Assume there are $\delta n = d_1 d_2/8$ errors, then there are at most $d_1/4$ codewords of the second code which are corrupted for at least $d_2/2$ bits. Since the distance of the second code is d_2, there are $n_1 - d_1/4$ blocks can be decoded to their correct messages.

Hence the expectation of the number of correctly recovered symbols of the first code is $\frac{n_1 - d_1/4}{n_1} 8k_1 \geq 6k_1$. By Chernoff Bound, with probability $1 - 2^{-\Theta(k_1)}$, there are at least $5k_1$ symbols that are correctly recovered. Note that if we look at the $8k_1$ queried blocks they also get a $(8k_1, k_1, 7k_1 + 1)$ Reed-Solomon code since it is the evaluation of the degree $k_1 - 1$ polynomial on $8k_1$ distinct values in the field $\mathbb{F}_2^{n_2}$. Thus, our recovered symbols form a string which has only distance $3k_1$ from a codeword of the code. We can get the correct message by using the decoding algorithm of the code.

□

In the discussion below, we will use the following error correcting code for edit distance:

$$\mathbb{C}_0 : \{0,1\}^N \to \{0,1\}^{5N}$$

such that for each message $S \in \{0,1\}^N$, $\mathbb{C}_0(S)$ is composed of two parts. The first part is a codeword $\mathbb{C}_0'(S)$ where $\mathbb{C}_0' : \{0,1\}^N \to \{0,1\}^{4N}$ is a greedily constructed code that can tolerate a constant fraction of edit errors guaranteed by Lemma 5.3.1. The second part is a buffer of 0's of length N. That is, $\mathbb{C}_0(S) = \mathbb{C}_0'(S) \circ 0^N$. We assume the normalized edit distance between any two codewords of $\mathbb{C}_0(S)$ is at least δ'.

Another key component in our construction against edit error is a searching algorithm developed by [97]. Their work provides an algorithm for searching from a weighted sorted list L with a constant fraction of errors. An element (i, a_i) in the sorted list is composed of two parts, an index i and content in that element, a_i. All elements in the list are sorted by their index, i.e. the j-th element in L is (j, a_j). Beyond that, each element is equipped with a non-negative weight. When sampling from the list, each element is sampled with probability proportional to its weight. In [97], the authors proved the following result.

Lemma 5.3.3. *[Theorem 16 of [97]] Assume L' is a corrupted version of a weighted sorted list L with k elements, such that the total weight fraction of corrupted elements is some constant δ of the total weight of L. And the weights have the property that all sequences for $r \geq 3$ elements in the list have total weight in the range $[r/2, 2r]$. Then, there is an algorithm for searching L. For at least a $1 - O(\delta)$ fraction of the original list's elements, it recovers them with probability at least $1 - \mathsf{neg}(k)$ for some negligible function neg. It makes a total of $O(\log^3 k)$ queries.*

This lemma enables us to search from a weighted sorted list (with corruption) with few $(\operatorname{polylog} k)$ queries. To make our proof self-contained, we will describe how to turn the encoded message in our construction into a weighted sorted list in the proof.

We are now ready to describe our construction.

Construction with Fixed Failure Probability

Shared Randomness

We first give the construction of an $(n, k = \Omega(n), \delta = O(1), q = \operatorname{polylog} k \log \frac{1}{\varepsilon}, \varepsilon)$ randomized LDC. As before, We start with the construction assuming shared randomness.

Construction 5.3.1. *We construct an $(n, k = \Omega(n), \delta = O(1), q = \operatorname{polylog} k \log \frac{1}{\varepsilon}, \varepsilon)$-LDC with randomized encoding.*

Let δ_0, γ_0 be some proper constants in $(0, 1)$.

Let $(\mathsf{Enc}_0, \mathsf{Dec}_0)$ be an asymptotically good (n_0, k_0, d_0) error correcting code for Hamming errors over alphabet $\{0,1\}^{\log k}$. Here we pick $n_0 = O(\log \frac{1}{\varepsilon})$, $k_0 = \gamma_0 n_0$, and $d = 2\delta_0 n_0 + 1$.

Let π be a random permutation. And $r_i \in \{0,1\}^{\log k}$ for $i \in [\frac{n_0 k}{k_0 \log k}]$ be $\frac{n_0 k}{k_0 \log k}$ random masks. Both π and r_i's are shared between the encoder and decoder.

Let $\mathbb{C}_0 : \{0,1\}^{2\log k} \to \{0,1\}^{10\log k}$ be the asymptotically good code for edit error described previously that can tolerate a δ' fraction of edit error.

The encoding function $\mathsf{Enc} : \{0,1\}^k \to \{0,1\}^n$ is a random function as follows

1. *On input $x \in \{0,1\}^k$, view x as a string over alphabet$\{0,1\}^{\log k}$of length $k/\log k$. We write $x = x_1x_2\cdots x_{k/\log k} \in (\{0,1\}^{\log k})^{k/\log k}$;*
2. *Divide x into small blocks of length k_0, s.t. $x = B_1^{(0)} \circ B_2^{(0)} \circ \cdots \circ B^{(0)}_{k/(k_0\log k)})$. Here, $B_i^{(0)} \in (\{0,1\}^{\log k})^{k_0}$ for $i \in [k/(k_0\log k)]$ is a concatenation of k_0 symbols in x;*
3. *Encode each block with Enc_0. Concatenate them to get $y^{(1)} = \mathsf{Enc}_0(B_1^{(0)}) \circ \mathsf{Enc}_0(B_2^{(0)}) \circ \cdots \circ \mathsf{Enc}_0(B^{(0)}_{k/(k_0\log k)})$. Notice that each $\mathsf{Enc}_0(B_1^{(0)})$ is composed of n_0 symbols in $\{0,1\}^{\log k}$. Write $y^{(1)}$ as a string over alphabet $\{0,1\}^{\log k}$, we have $y^{(1)} = B_1^{(1)} \circ B_2^{(1)} \circ \cdots \circ B^{(1)}_{\frac{n_0k}{k_0\log k}}$ such that $B_i^{(1)} \in \{0,1\}^{\log k}$;*
4. *Let $N = \frac{n_0k}{k_0\log k}$. Permute these N symbols of $y^{(1)}$ with permutation π to get $y^{(2)} = B_1^{(2)} \circ B_2^{(2)} \circ \cdots \circ B_N^{(2)}$ such that $B^{(2)}_{\pi(i)} = B_i^{(1)}$;*
5. *Let $b_i \in \{0,1\}^{\log k}$ be the binary representation of $i \in [N]$. This is fine since $N < k$ when k is larger enough. We compute $B_i^{(3)} = \mathbb{C}_0(b_i \circ (B_i^{(2)} \oplus r_i)) \in \{0,1\}^{10\log k}$ for each $i \in [T]$. We get $y = B_1^{(3)} \circ B_2^{(3)} \circ \cdots \circ B_N^{(3)} \in (\{0,1\}^{10\log k})^N$;*
6. *Output y.*

The decoding function $\mathsf{Dec} : [k] \times \{0,1\}^n \to \{0,1\}$ takes two inputs, an index $i_0 \in [k]$ of the message bit the decoder wants to know and $\omega \in \{0,1\}^n$, the received (possibly corrupted) codeword. It proceeds as follows

1. *On input index i_0 and the received codeword ω. We assume the i_0-th bit lies in $B_i^{(0)}$, i.e. the i-th block of x;*

2. *Notice that* $\mathsf{Enc}_0(B_i^{(0)}) = B_{(i-1)n_0+1}^{(1)} \circ B_{(i-1)n_0+2}^{(1)} \circ \cdots \circ B_{in_0}^{(1)}$. *For each* $j \in \{(i-1)n_0+1, (i-1)n_0+2, \ldots, in_0\}$, *search from* y *to find the block* $B_{\pi(j)}^{(3)}$ *using the algorithm from [97]. Then we can get a possibly corrupted version of* $\mathsf{Enc}_0(B_i^{(0)})$

3. *Run the decoding algorithm* Dec_0 *to find out* $B_i^{(0)}$. *This gives us the* i_0*-th bit of* x.

Lemma 5.3.4. *The above construction 5.3.1 gives an efficient* $(n, k = \Omega(n), \delta = O(1), q = \operatorname{polylog} k \log \frac{1}{\varepsilon}, \varepsilon)$ *randomized LDC against edit error.*

Proof. We first show both the encoding and decoding can be done in polynomial time. Although for the ease of description, we picked the greedily constructed code $\mathbb{C}_0$, which can be inefficient. The codeword size of $\mathbb{C}_0$ is $O(\log k)$. Decoding one block can be finished in time polynomial in k. And for our purpose, we need to encode $O(k/\log k)$ blocks and decode $\operatorname{polylog} n \log \frac{1}{\varepsilon}$ blocks. Thus, the additional time caused by this layer of encoding is polynomial in n. The rest part also runs in polynomial time because generating random permutation, permuting bits, and the code $(\mathsf{Enc}_0, \mathsf{Dec}_0)$ are all in polynomial time. Thus, our code is polynomial time. We note that $\mathbb{C}_0$ can be replaced by an efficient code for edit error.

We use the same notation as in our construction. Let $y = B_1^{(3)} \circ B_2^{(3)} \circ \cdots \circ B_N^{(3)}$ be the correct codeword. We denote the length of each block $B_j^{(3)}$ by $b = 10 \log k$ and if we view y as a binary string, the length of y is $N' (= Nb)$, which is $O(k)$. We call the received (possibly corrupted) codeword ω. Since we can always truncate (or pad with 0) to make the length of ω be N', which will only increase the edit distance by a factor no more than 2. Without loss of generality, we assume the length of ω is also N'.

The decoding function Dec has two inputs: an index $i_0 \in [k]$ of the message bit the decoder wants to see and the received codeword ω. The decoder also has access to the shared randomness, which has two parts: a permutation π and N

random masks, each of length $\log k$. The first step is to figure out the indices of the blocks it wants to query by using the shared permutation π. Then, we can query these blocks one by one. We assume the i_0-th bit lies in $B_i^{(0)}$. Then, notice that $\mathsf{Enc}_0(B_i^{(0)}) = B^{(1)}_{(i-1)n_0+1} \circ B^{(1)}_{(i-1)n_0+2} \circ \cdots \circ B^{(1)}_{in_0}$. Our goal is to find the block $B^{(3)}_{\pi(j)}$ for each $j \in \{(i-1)n_0+1, (i-1)n_0+2, \ldots, in_0\}$.

One thing we need to clarify is how to query a block since we do not know the starting point of each block in the corrupted codeword. This is where we use the techniques developed by [97]. In the following, assume we want to find the block $B_i^{(3)}$.

We can view y as a weighted sorted list L of length N such that the i-th element in L is simply $B_i^{(3)}$ with weight b. We show that ω can be viewed as a corrupted list L'. There is a match from y to ω which can be described by a function $f : [N] \to [N] \cup \{\perp\}$. If the i-th bit is preserved after the edit error, then $f(i) = j$ where j is the position of that particular bit in ω. If the i-th bit is deleted, then $f(i) = \perp$.

We say the i-th bit in y is preserved after the corruption if $f(i) \neq \perp$. For each block $B_i^{(3)}$ that is not completely deleted, let $v_i = f(u_i)$, such that u_i is the index of the first preserved bit in the block $B_i^{(3)}$. We say the block $B_i^{(3)}$ is *recoverable* if $\Delta_E(B_i^{(3)}, \omega_{[v'_i, v'_i+b-1]}) \leq \delta'$ for some index v'_i that is at most $\delta' b$ away from v_i.

Claim 5.3.1. *If $\Delta_E(\omega, y) \leq \delta$, each block is recoverable with probability at least $1 - \delta/\delta'$.*

Proof. The code $\mathbb{C}_0$ is greedily constructed and resilient to δ' fraction of error. Here, δ is picked smaller than δ'. To make an block corrupted, the adversary needs to produce at least a δ' fraction of edit error in that block. The adversary channel can corrupt at most $\frac{\delta}{\delta'}$ fraction of all blocks. □

The searching algorithm requires sampling some elements from the sorted list L' corresponding to the corrupted codeword ω. We now explain how to do the sampling. We start by first randomly sample a position r and read a substring of $2r+1$ bits

$\omega_{[r-b,r+b]}$ from ω. Then, we try each substring in $\omega_{[r-b,r+b]}$ of length b from left to right until we find first substring that is δ_0-close to some codeword of $\mathbb{C}$ under the normalized edit distance. If we did find such a substring, we decode it and get $b_j \circ B_j^{(2)} \oplus r_j$. Otherwise, output a special symbol $\perp$.

We will regard consecutive intervals in ω as elements in the weighted sorted list where weight of the element is simply the length of the corresponding interval. The correspondence can be described as following.

For a recoverable block $B_i^{(3)}$, again, we let $v_i = f(u_i)$, such that u_i is the index of the first preserved character in block $B_i^{(3)}$. We want to find an interval I_i in ω to represent $B_i^{(3)}$ in the list L'. Since $B_i^{(3)}$ is a codeword of $\mathbb{C}_0$ of length b. Every interval in its first $4/5$ part has at least half of 1's and the last $1/5$ part are all 0's. For any $v \in [v_i - b + 2\delta' b, v_i - 2\delta' b]$, we know $\omega_{[v,v+b-1]}$ can not be δ' close to any codeword in $\mathbb{C}_0$. It is because the last $1/5$ part of $\omega_{[v,v+b-1]}$ contains at least $\delta' b$ 1's. Thus, in the sampling procedure, if $r \in [v_i + 2\delta' b, v_i + b]$, it will return $B_i^{(3)}$. The length of I_i is at least $1 - 2\delta' b$. Let I_i be the maximal inteval containing $[v_i + 2\delta' b, v_i + b]$, such that, if sampling r in I_i, it will output same codeword in $\mathbb{C}$. Also, we argue the length of the inteval I_i is no larger than $1 + 4\delta'$. Since if $r \geq v_i + (1 + 2\delta')b$ or $r \leq v_i - 2\delta' b$, any substring of length b in $[r - b, r + b]$ is at least be δ' far from $B_i^{(3)}$ and thus output a different codeword. We note due to the existence of adversary insertion, it is possible to get $B_i^{(3)}$ outside of I_i. But this does not affect our analysis below.

The above method for sampling a block from ω gives us a natural way to interpret ω as a corrupted weighted list L'. The length of ω is equal to the sum of weights of all its elements. For each recoverable block, the interval I_i as described above is an element with weight equal to its length. We note that all I_i's are disjoint. We consider the remaining characters in ω as corrupted elements in the list L. For those elements, the sampling algorithm will output wrong result or $\perp$. Since the interval of the remaining elements can be large, we divide such interval into small intervals each has length no larger than b and consider each small intervals as an element.

Next, we argue the weight fraction of corrupted elements is small. Due to Claim 5.3.1, there are at least a $1-\delta/\delta'$ fraction of blocks recoverable after the corruption. For each recoverable block $B_i^{(3)}$, we can find a interval I_i in ω with length at least $(1-2\delta_0)b$. Thus, the total weight of uncorrupted elements from the original list L is at least $(1-\delta/\delta')(1-2\delta')N = (1-2\delta-2\delta_0-\delta/\delta_0)N$. Let $\delta_1 = 2\delta+2\delta'+\delta/\delta' = O(\delta)$, we know the total weight of corrupted elements is at most δ_1 fraction of the total weight of list L'. We assume δ_1 is small by properly picking δ and δ'.

By Lemma 5.3.3, $1-\delta_2$ fraction of elements can be decoded correctly with high probability $(1-\mathsf{neg}(n))$ for some constant $\delta_2 = O(\delta)$. Since the content of each block is protected by random masks. The adversary can learn nothing about the random permutation π used for encoding. Thus, we can assume each block is recoverable with same probability. We want to search from y to find all blocks $B_{\pi(j)}^{(3)}$ for each $j \in \{(i-1)n_0+1, (i-1)n_0+2, \dots, in_0\}$ to get a possibly corrupted version of $\mathsf{Enc}_0(B_i^{(0)})$.

By Lemma 5.1.3, with probability at least $1-2^{-0.01n_0/3}$, the number of blocks in $\mathsf{Enc}_0(B_i^{(0)})$ that are not decodable is at most $1.1\delta_2 n_0$. By choosing n_0 to be large enough and δ_0 to be small enough, Dec_0 can recover the desired block $\mathsf{Enc}_0(B_i^{(0)})$ with probability at least $1-\varepsilon$.

Finally, we count the number of queries made. Searching one block queries polylog k symbols from the corrupted codeword ω. We need to search $n_0 = O(\log\frac{1}{\varepsilon})$ blocks. The total number of queries made is $\operatorname{polylog} k \log\frac{1}{\varepsilon}$. □

Oblivious Channel

Now, we consider the oblivious channel. In this model, we assume the adversary can not read the codeword. The key idea is to encode a description of the permutation in the codeword. The decoder can then use the description to recover the permutation π which is used to encode the message. We can use the binary error correcting code from Lemma 5.3.2 to encode the description.

We now give the construction.

Construction 5.3.2. *We construct an* $(n, k = \Omega(n), \delta = O(1), q = \operatorname{polylog} k \log \frac{1}{\varepsilon}, \varepsilon)$*-LDC with randomized encoding against the oblivious channel model.*

Let δ_0*,* γ_0 *be some proper constants in* $(0, 1)$*.*

Let $(\mathsf{Enc}_0, \mathsf{Dec}_0)$ *be an asymptotically good* (n_0, k_0, d_0) *error correcting code for Hamming errors on alphabet set* $\Sigma = \{0, 1\}^{10 \log k}$*. Here we pick* $n_0 = O(\log \frac{1}{\varepsilon})$*,* $k_0 = \gamma_0 n_0$*, and* $d = 2\delta_0 n_0 + 1$*.*

Let $\mathbb{C}_0 : \{0, 1\}^{2 \log k} \to \{0, 1\}^{10 \log k}$ *be the asymptotically good code for edit error as described above that can tolerate a* δ' *fraction of edit error.*

The encoding function $\mathsf{Enc} : \{0, 1\}^k \to \{0, 1\}^n$ *is a random function as follows:*

1. *Encode the message* x *with Construction 5.3.1 without doing steps 4,5, and 6. This does not require knowing permutation* π*. View the sequence we get as a binary string. View the output as a binary string* y *with length* $n/10$*;*

2. *Let* $N_1 = \frac{n}{20 \log k}$*. Generate a random seed* $r \in \{0, 1\}^d$ *of lengh* d*. Here,* $d = O(\kappa \log N_1 + \log(1/\varepsilon_\pi)) = O(\log n \log \frac{1}{\varepsilon})$*. Use* r *to sample a* $\varepsilon_\pi = \varepsilon/10$*-almost* $\kappa = O(\log \frac{1}{\varepsilon})$*-wise independent random permutation* $\pi : [N_1] \to [N_1]$ *using Theorem 5.7;*

3. *Let* δ_1 *be a properly chosen constant larger than* δ*. Encoding* r *with an* $(n/20, d, \delta_1 n)$ *error correcting code from Lemma 5.3.2, we get* $z \in \{0, 1\}^{n/10}$*;*

4. *View* $y \circ z$ *as a sequence in* $\{0, 1\}^{n/10}$*. Divide* $y \circ z \in \{0, 1\}^{n/10}$ *into small blocks of size* $\log k$*. Write* $y \circ z$ *as* $y \circ z = B_1 \circ B_2 \circ \cdots \circ B_{n/(10 \log k)}$*;*

5. *Permute first half of blocks with random permutation π to get $u' = B_1^{(1)} \circ B_2^{(1)} \circ \cdots \circ B_{n/(10\log k)}^{(1)}$ such that $B_{\pi(i)}^{(1)} = B_i$ for $i \le n/(20\log k)$ and $B_i^{(1)} = B_i$ for i larger than $n/(20\log k)$;*

6. *Let b_i be the binary representation of i, for each $i \in n/(10\log k)$, encode $b_i \circ B_i^{(1)}$ with code $\mathbb{C}_0$ to get $B_i^{(2)}$;*

7. *Output $u = B_1^{(2)} \circ B_2^{(2)} \circ \cdots \circ B_{n/(10\log k)}^{(2)}$.*

The decoding function $\mathsf{Dec} : [k] \times \{0,1\}^n \to \{0,1\}$ takes two inputs, an index $i_0 \in [k]$ of the message bit the decoder wants to know and $\omega \in \{0,1\}^n$, the received (possibly corrupted) codeword. It proceeds as follows:

1. *Search for at most $O(\log n \log \frac{1}{\varepsilon})$ blocks to decode the random seed r;*

2. *Use r to generate the random permutation π;*

3. *Run the same decoding algorithm as in the Construction 5.3.1 on the first half of ω to decode x_{i_0}.*

Lemma 5.3.5. *The above construction 5.3.1 gives an efficient $(n, k = \Omega(n), \delta = O(1), q = \mathrm{polylog}(n)\log\frac{1}{\varepsilon}, \varepsilon)$ randomized LDC against an oblivious channel with edit error.*

Proof. The proof of efficiency of this construction follows directly from the proof of Lemma 5.3.4.

We denote the uncorrupted codeword by u and the received codeword by ω. We assume $\Delta_E(u,\omega) \le \delta$. We first divide it into two parts ω_1 and ω_2 each of equal length such that $\omega = \omega_1 \circ \omega_2$. Similarly, we have $u = u_1 \circ u_2$ with $|u_1| = |u_2| = n/2$. We have $\Delta_E(u_1, \omega_1) \le \delta$ and $\Delta_E(u_2, \omega_2) \le \delta$.

The first step is to decode r. By Lemma 5.3.2, we need to query $O(d)$ bits of z to decode r with high probability $1 - \mathsf{neg}(k)$. Thus, we need to query $O(\log k \log \frac{1}{\varepsilon})$

blocks in the second half of ω. We can then use the same searching algorithm described in the proof of Lemma 5.3.4 to query each block. Since we randomly choose blocks to query, we can recover r with probability $1 - \mathsf{neg}(k)$ by making $\operatorname{polylog} k \log \frac{1}{\varepsilon}$ queries to ω_2.

Once we know r, we can use it to generate the random permutation π. We run the decoding algorithm from Construction 5.3.1 with input i_0 and ω_1. The remaining part is the same as the proof of Lemma 5.3.4. Notice that recovering r takes $\operatorname{polylog} k \log \frac{1}{\varepsilon}$ queries. The query complexity for our code is still $\operatorname{polylog} k \log \frac{1}{\varepsilon}$. □

Construction with Flexible Failure Probability

Shared Randomness

We now give the construction for flexible failure peobability against edit error.

Construction 5.3.3. *We construct an $(n, k = \Theta(n/\log n), \delta = O(1))$ randomized LDC with query complexity function $q = \operatorname{polylog} k \log \frac{1}{\varepsilon}$ for any given failure probability ε.*

Let π be a random permutation. And $r_i \in \{0,1\}^{\log k}$ for each $i \in n/(10 \log k)$ be random masks. Both π and r_i's are shared between the encoder and decoder.

With properly picked constant γ_0 and δ_0. Let $(\mathsf{Enc}_0, \mathsf{Dec}_0)$ be a (n_0, k_0, d_0) error correcting code on alphabet set $\{0,1\}^{\log k}$. Here, $n_0 = \gamma_0^{-1} \log n$, $k_0 = \log n$, $d_0 = 2\delta_0 n_0 + 1$, for constant $\gamma_0, \delta_0 \in [0,1]$.

For each $i \in [\log n]$, let Enc_i be the encoding algorithm of an $(n_i, k, \delta_i = O(1), q_i, \varepsilon = 2^{-2^i})$ randomized LDC from Construction 5.3.1 without steps 4,5, and 6. Let Dec_i be the correponding decoding algotithm.

Let $\mathbb{C}_1 : \{0,1\}^{3 \log k} \to \{0,1\}^{15 \log k}$ be the asymptotically good code for edit error as described above that can tolerate a δ' fraction of edit error.

Encoding function $\mathsf{Enc} : \{0,1\}^{k = \Omega(n)} \to \{0,1\}^n$ is as follows.

1. *On input $x \in \{0,1\}^k$, compute $y_i = \mathsf{Enc}_i(x)$ for every $i \in [\log n]$. As before, we view $y_i = B_1^i \circ B_2^i \circ \cdots \circ B_N^i$ as the concatenation of N symbols over alphabet $\{0,1\}^{\log k}$;*

2. *Let M be a $\log n \times N$ matrix such that $M[i][j] \in \{0,1\}^{\log k}$ is the j-th symbol of y_i;*

3. *Let M_j be the j-th column of M for each $j \in [N]$. We compute $z_j = \mathsf{Enc}_0(M_j)$. Notice that $z_j = \mathsf{Enc}_0(M_j) \in (\{0,1\}^{\log k})^{n_0}$ is a string of length n_0 over alphabet $\{0,1\}^{\log k}$;*

4. *Let $y^{(0)}$ be the concatenation of z_j's for $j \in [N]$. Then $y^{(0)} = z_1 \circ z_2 \cdots \circ z_N$ is a string of length n_0N over alphabet $\{0,1\}^{\log k}$. Let $n = 15\log(k)n_0N = O(k \log k)$;*

5. *Permute the symbols of $y^{(0)}$ with permutation π to get $y^{(1)} = B_1^{(1)} \circ B_2^{(1)} \circ \cdots \circ B_{n_0N}^{(1)}$ such that $B_{\pi(i)}^{(1)} = B_i^{(0)}$;*

6. *Since $n_0N = O(k)$, we assume $n_0N = ck$ for some constant c. When k is large enough, we assume $ck < k^2$. Let $b_i \in \{0,1\}^{2\log k}$ be the binary representation of i for each $i \in [n_0N]$. We compute $B_i^{(2)} = \mathbb{C}_1(b_i \circ (B_i^{(1)} \oplus r_i)) \in \{0,1\}^{10\log k}$ for each $i \in [n_0N]$. We get $y = B_1^{(2)} \circ B_2^{(2)} \circ \cdots \circ B_{N'}^{(2)}) \in (\{0,1\}^{15\log k})^{n_0N}$;*

7. *Output y as a binary string of length n.*

Decoding function Dec is a randomized algorithm takes three inputs: an index of the bit the decoder wants to see,the received codeword ω, and the desired failure probability ε. It can be described as follows.

1. *On input i_0, ω, ε, find the smallest i such that $2^{-2^i} \leq \varepsilon$. If it cannot be found, then query the whole ω;*

2. *Compute $w = \mathsf{Dec}_i(i_0, y_i)$ but whenever Dec_i wants to query an j-th symbol of y_i, we decode z_j. Notice that $z_j = \mathsf{Enc}_0(M_j)$ contains n_0 symbols over alphabet*

$\{0,1\}^{\log k}$. Each symbol is encoded in a block in y. The indices of these blocks can be recovered by π. We search each of these n_0 blocks in ω. We can then get the j-th block of y_i from $\mathsf{Dec}_0(z_j)$;

3. *Output w.*

Lemma 5.3.6. *The above construcion 5.3.3 gives an efficient $(n, k = \Theta(n/\log n), \delta = O(1))$ randomized LDC that can recover any bit with probability at least $1-\varepsilon$ for any constant $\varepsilon \in (0,1)$, with query complexity $q = \operatorname{polylog} k \log \frac{1}{\varepsilon}$.*

Proof. In the encoding, we need to encode $n_0 N = O(n/\log n)$ blocks with edit error code $\mathbb{C}_0$. And in the decoding, we need decode at most $\operatorname{polylog} n \log \frac{1}{\varepsilon}$ blocks. The time caused by the second layer is polynomial in n. Thus, our construction is efficient.

After a δ fraction of edit error. There is some constant $\delta_1 = O(\delta)$ such tath for at least $1-\delta_1$ fraction of blocks, we can use the searching algorithm to find and decode them correctly with probability $1-\mathsf{neg}(n)$. Thus, as is described in the construction, whenever we want to query the j-th symbol of y_i, we fist decode z_j. The symbols of z_j is encoded in n_0 blocks in ω. With π, we know the indices of these n_0 blocks. We can than perform the search as described in the proof of Lemma 5.3.4. By picking proper n_0, the decoding of z_j fails with a small probability. The rest of analysis follows directly from the proof of Lemma 5.3.4.

For the query complexity, we need to query $O(n_0 \log \frac{1}{\varepsilon})$ blocks. Since $n_0 = O(\log n) = O(\log k)$, the total number of queries made is still $\operatorname{polylog} k \log \frac{1}{\varepsilon}$. □

Oblivious Channel

Our construction for flexible failure probability against oblivious channel combines Construction 5.3.2 and Construction 5.3.3. More specifically, following Construction 5.3.3, we replace the encoding functions Enc_i for each $i \in [\log n]$ from Construction

5.3.1 with the encoding functions from Constructon 5.3.2. The analysis follows directly. We omit the details.

5.4 Open Problems

We list some of the open problems below.

1. **Better lower bounds and/or explicit constructions** While our lower bounds here are exponential in the message length, it is still conceivable that such Insdel LDCs do not actually exist, a conjecture raised in [98]. If on the other hand such codes do exist, it would be very intriguing to have better lower bounds and/or have explicit constructions, even for $q = O(\log n/\log\log n)$. It appears to be highly non-trivial to construct such codes, which is in contrast to Hamming LDCs, where the simple Hadamard code is a classic example of a constant-query (in fact 2-query) LDC.

2. **Relaxed Insdel LDCs/LCCs** Relaxed (Hamming) LDCs/LCCs are variants in which the decoder is allowed some small probability of outputing a "don't know" answer, while it should answer with the correctly decoded bit most of the time. [128] proposed these variants and gave constructions with constant query complexity and codeword length $m = n^{1+\varepsilon}$. More recently [129] extended the notion to LCCs, and proved similar bounds, which are tight [130]. An open problem here is to understand tight bounds for the relaxed Insdel variants of LDCs/LCCs.

3. **Larger alphabet size** We believe our proofs generalize to larger alphabet sizes, and leave the precise bounds in terms of the alphabet size as an open problem. All the above directions may also be asked for larger alphabet sizes.

4. **LDCs with randomized encoding** Can we achieve better constructions of

LDCs with randomized encoding? Some interesting directions to explore include: constructions of LDCs with randomized encoding such that the decoder can succeed without knowing the randomess of encoding, constructions achieving flexible failure probability with constant rate, and constructions with improved number of queries for an oblivious channel. Another interesting question is, if the adversary is allowed to know the randomness used by the encoding, can we still achieve much better rate-query tradeoffs by using LDCs with randomized encoding?

References

1. Banal, J. L. *et al.* Random access DNA memory using Boolean search in an archival file storage system. *Nature Materials* **20,** 1272–1280 (9 2021).
2. Backurs, A. & Indyk, P. *Edit distance cannot be computed in strongly subquadratic time (unless SETH is false)* in *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing* (2015).
3. Abboud, A., Backurs, A. & Williams, V. V. *Tight hardness results for LCS and other sequence similarity measures* in *Proceedings of the Fifty-Sixth IEEE Annual Symposium on Foundations of Computer Science* (2015).
4. Impagliazzo, R., Paturi, R. & Zane, F. Which problems have strongly exponential complexity. *Journal of Computer and System Sciences* **63,** 512–530 (2001).
5. Charkraborty, D., Das, D., Goldenberg, E., Koucky, M. & Saks, M. *Approximating Edit Distance Within Constant Factor in Truly Sub-Quadratic Time* in *Proceedings of the Fifty-Ninth Annual IEEE Symposium on Foundations of Computer Science* (2018).
6. Brakensiek, J. & Rubinstein, A. *Constant-factor approximation of near-linear edit distance in near-linear time* in *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020* (2020), 685–698.
7. Koucký, M. & Saks, M. E. *Constant factor approximations to edit distance on far input pairs in nearly linear time* in *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020* (2020), 699–712.
8. Andoni, A. & Nosatzki, N. S. *Edit Distance in Near-Linear Time: it's a Constant Factor* in *Proceedings of the 61st Annual Symposium on Foundations of Computer Science (FOCS)* (2020).
9. Hajiaghayi, M., Seddighin, M., Seddighin, S. & Sun, X. *Approximating lcs in linear time: Beating the barrier* in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms* (2019), 1181–1200.
10. Rubinstein, A., Seddighin, S., Song, Z. & Sun, X. *Approximation Algorithms for LCS and LIS with Truly Improved Running Times* in *Foundations of Computer Science (FOCS), 2019 IEEE 60th Annual Symposium on* (2019).
11. Rubinstein, A. & Song, Z. *Reducing approximate longest common subsequence to approximate edit distance* in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2020), 1591–1600.

12. Chakraborty, D., Goldenberg, E. & Koucký, M. *Low Distortion Embedding from Edit to Hamming Distance using Coupling* in *Proceedings of the 48th IEEE Annual Annual ACM SIGACT Symposium on Theory of Computing* (2016).

13. Belazzougui, D. & Zhang, Q. *Edit distance: Sketching, streaming, and document exchange* in *FOCS* (2016).

14. Gopalan, P., Jayram, T. S., Krauthgamer, R. & Kumar, R. *Estimating the sortedness of a data stream* in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007* (eds Bansal, N., Pruhs, K. & Stein, C.) (SIAM, 2007), 318–327.

15. Kiyomi, M., Ono, H., Otachi, Y., Schweitzer, P. & Tarui, J. Space-efficient algorithms for longest increasing subsequence. *Theory of Computing Systems,* 1–20 (2018).

16. Savitch, W. J. Relationships between nondeterministic and deterministic tape complexities. *Journal of computer and system sciences* **4,** 177–192 (1970).

17. Cheng, K., Jin, Z., Li, X. & Zheng, Y. Space efficient deterministic approximation of string measures. *arXiv preprint arXiv:2002.08498* (2020).

18. Cheng, K. *et al. Streaming and small space approximation algorithms for edit distance and longest common subsequence* in *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)* (2021).

19. Leiserson, C. E., Rivest, R. L., Cormen, T. H. & Stein, C. *Introduction to algorithms* (MIT press Cambridge, MA, 2001).

20. Grabowski, S. New tabuation and sparse dynamic programming based techniques for sequence similarity problems. *Discrete Applied Mathematics* **212,** 96–103 (2016).

21. Bringmann, K. & Künnemann, M. *Quadratic conditional lower bounds for string problems and dynamic time warping* in *2015 IEEE 56th Annual Symposium on Foundations of Computer Science* (2015), 79–97.

22. Abboud, A. & Rubinstein, A. *Fast and deterministic constant factor approximation algorithms for LCS imply new circuit lower bounds* in *9th Innovations in Theoretical Computer Science Conference (ITCS 2018)* (2018).

23. Landau, G. M., Myers, E. W. & Schmidt, J. P. Incremental string comparison. *SIAM Journal on Computing* **27,** 557–582 (1998).

24. Bar-Yossef, Z., Jayram, T., Krauthgamer, R. & Kumar, R. *Approximating edit distance efficiently* in *FOCS* (2004).

25. Batu, T., Ergun, F. & Sahinalp, C. *Oblivious string embeddings and edit distance approximations* in *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm* (2006).

26. Andoni, A. & Onak, K. Approximating edit distance in near-linear time. *SIAM Journal on Computing* **41,** 1635–1648 (2012).

27. Abboud, A. & Backurs, A. *Towards hardness of approximation for polynomial time problems* in *8th Innovations in Theoretical Computer Science Conference (ITCS 2017)* (2017).

28. Chen, L., Goldwasser, S., Lyu, K., Rothblum, G. N. & Rubinstein, A. *Fine-grained complexity meets IP= PSPACE* in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms* (2019), 1–20.

29. Aldous, D. & Diaconis, P. Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem. *Bulletin of the American Mathematical Society* **36,** 413–432 (1999).

30. Fredman, M. L. On computing the length of longest increasing subsequences. *Discrete Mathematics* **11,** 29–35 (1975).

31. Saks, M. & Seshadhri, C. *Estimating the longest increasing sequence in polylogarithmic time* in *Proceedings of the Fifty-First Annual IEEE Symposium on Foundations of Computer Science* (2010).

32. Naumovitz, T. & Saks, M. *A polylogarithmic space deterministic streaming algorithm for approximating distance to monotonicity* in *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms* (2014), 1252–1262.

33. Kiyomi, M., Horiyama, T. & Otachi, Y. Longest common subsequence in sublinear space. *Information Processing Letters* **168,** 106084 (2021).

34. Hajiaghayi, M., Seddighin, S. & Sun, X. *Massively parallel approximation algorithms for edit distance and longest common subsequence* in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms* (2019), 1654–1672.

35. Boroujeni, M., Ghodsi, M. & Seddighin, S. Improved MPC algorithms for edit distance and Ulam distance. *IEEE Transactions on Parallel and Distributed Systems* **32,** 2764–2776 (2021).

36. Liben-Nowell, D., Vee, E. & Zhu, A. *Finding longest increasing and common subsequences in streaming data* in *International Computing and Combinatorics Conference* (2005), 263–272.

37. Sun, X. & Woodruff, D. P. *The communication and streaming complexity of computing the longest common and increasing subsequences* in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms* (2007), 336–345.

38. Andoni, A., Krauthgamer, R. & Onak, K. *Polylogarithmic approximation for edit distance and the asymmetric query complexity* in *2010 IEEE 51st Annual Symposium on Foundations of Computer Science* (2010), 377–386.

39. Saks, M. & Seshadhri, C. *Space efficient streaming algorithms for the distance to monotonicity and asymmetric edit distance* in *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms* (2013), 1698–1709.

40. Saha, B. *Fast & Space-Efficient Approximations of Language Edit Distance and RNA Folding: An Amnesic Dynamic Programming Approach* in *FOCS* (2017).

41. Li, X. & Zheng, Y. *Lower Bounds and Improved Algorithms for Asymmetric Streaming Edit Distance and Longest Common Subsequence* in *41st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2021, December 15-17, 2021, Virtual Conference* (eds Bojanczyk, M. & Chekuri, C.) **213** (Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021), 27:1–27:23.

42. Gál, A. & Gopalan, P. Lower bounds on streaming algorithms for approximating the length of the longest increasing subsequence. *SIAM Journal on Computing* **39,** 3463–3479 (2010).

43. Ergun, F. & Jowhari, H. *On distance to monotonicity and longest increasing subsequence of a data stream* in *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms* (2008), 730–736.

44. Chakrabarti, A. A note on randomized streaming space bounds for the longest increasing subsequence problem. *Information Processing Letters* **112,** 261–263 (2012).

45. Levenshtein, V. I. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* **10.** Doklady Akademii Nauk SSSR, V163 No4 845-848 1965, 707–710 (1966).

46. Katz, J. & Trevisan, L. *On the efficiency of local decoding procedures for error-correcting codes* in *STOC* (2000), 80–86.

47. Sudan, M., Trevisan, L. & Vadhan, S. P. *Pseudorandom Generators without the XOR Lemma (Abstract)* in *CCC* (1999), 4.

48. Babai, L., Fortnow, L., Levin, L. A. & Szegedy, M. *Checking Computations in Polylogarithmic Time* in *STOC* (1991), 21–31.

49. Lund, C., Fortnow, L., Karloff, H. J. & Nisan, N. Algebraic Methods for Interactive Proof Systems. *J. ACM* **39,** 859–868 (1992).

50. Blum, M., Luby, M. & Rubinfeld, R. Self-Testing/Correcting with Applications to Numerical Problems. *J. Comput. Syst. Sci.* **47,** 549–595 (1993).

51. Blum, M. & Kannan, S. Designing Programs that Check Their Work. *J. ACM* **42,** 269–291 (1995).

52. Chor, B., Kushilevitz, E., Goldreich, O. & Sudan, M. Private Information Retrieval. *J. ACM* **45,** 965–981 (1998).

53. Chen, V., Grigorescu, E. & de Wolf, R. Error-Correcting Data Structures. *SIAM J. Comput.* **42,** 84–111 (2013).

54. Andoni, A., Laarhoven, T., Razenshteyn, I. P. & Waingarten, E. *Optimal Hashing-based Time-Space Trade-offs for Approximate Near Neighbors* in *SODA* (2017), 47–66.

55. Trevisan, L. Some Applications of Coding Theory in Computational Complexity. *CoRR* **cs.CC/0409044** (2004).

56. Gasarch, W. I. A Survey on Private Information Retrieval (Column: Computational Complexity). *Bulletin of the EATCS* **82,** 72–107 (2004).

57. Kerenidis, I. & de Wolf, R. Exponential lower bound for 2-query locally decodable codes via a quantum argument. *J. Comput. Syst. Sci.* **69,** 395–420 (2004).

58. Wehner, S. & de Wolf, R. *Improved Lower Bounds for Locally Decodable Codes and Private Information Retrieval* in *ICALP* **3580** (Springer, 2005), 1424–1436.

59. Goldreich, O., Karloff, H. J., Schulman, L. J. & Trevisan, L. Lower bounds for linear locally decodable codes and private information retrieval. *Comput. Complex.* **15,** 263–296 (2006).

60. Woodruff, D. P. *New Lower Bounds for General Locally Decodable Codes.* tech. rep. (Weizmann Institute of Science, Israel, 2007).

61. Yekhanin, S. Towards 3-query locally decodable codes of subexponential length. *J. ACM* **55,** 1:1–1:16 (2008).

62. Yekhanin, S. Locally Decodable Codes. *Foundations and Trends in Theoretical Computer Science* **6,** 139–255 (2012).

63. Dvir, Z., Gopalan, P. & Yekhanin, S. Matching Vector Codes. *SIAM J. Comput.* **40,** 1154–1178 (2011).

64. Efremenko, K. 3-Query Locally Decodable Codes of Subexponential Length. *SIAM J. Comput.* **41,** 1694–1703 (2012).

65. Gál, A. & Mills, A. Three-Query Locally Decodable Codes with Higher Correctness Require Exponential Length. *ACM Trans. Comput. Theory* **3,** 5:1–5:34 (2012).

66. Bhattacharyya, A., Dvir, Z., Saraf, S. & Shpilka, A. Tight lower bounds for linear 2-query LCCs over finite fields. *Comb.* **36,** 1–36 (2016).

67. Bhattacharyya, A. & Gopi, S. Lower Bounds for Constant Query Affine-Invariant LCCs and LTCs. *ACM Trans. Comput. Theory* **9,** 7:1–7:17 (2017).

68. Dvir, Z., Saraf, S. & Wigderson, A. Superquadratic Lower Bound for 3-Query Locally Correctable Codes over the Reals. *Theory Comput.* **13,** 1–36 (2017).

69. Kopparty, S., Meir, O., Ron-Zewi, N. & Saraf, S. High-Rate Locally Correctable and Locally Testable Codes with Sub-Polynomial Query Complexity. *J. ACM* **64,** 11:1–11:42 (2017).

70. Bhattacharyya, A., Chandran, L. S. & Ghoshal, S. *Combinatorial Lower Bounds for 3-Query LDCs* in *ITCS* **151** (Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020), 85:1–85:8.

71. Kopparty, S. & Saraf, S. Guest Column: Local Testing and Decoding of High-Rate Error-Correcting Codes. *SIGACT News* **47,** 46–66 (2016).

72. Woodruff, D. P. A Quadratic Lower Bound for Three-Query Linear Locally Decodable Codes over Any Field. *J. Comput. Sci. Technol.* **27,** 678–686 (2012).

73. Ben-Aroya, A., Regev, O. & de Wolf, R. *A Hypercontractive Inequality for Matrix-Valued Functions with Applications to Quantum Computing and LDCs* in *FOCS* (IEEE Computer Society, 2008), 477–486.

74. Kopparty, S., Saraf, S. & Yekhanin, S. High-rate codes with sublinear-time decoding. *J. ACM* **61,** 28:1–28:20 (2014).

75. Schulman, L. J. & Zuckerman, D. Asymptotically good codes correcting insertions, deletions, and transpositions. *IEEE Transactions on Information Theory* **45,** 2552–2557 (1999).

76. Expected length of the longest common subsequence for large alphabets. *Advances in Mathematics* **197,** 480–498 (2005).

77. Guruswami, V. & Wang, C. Deletion codes in the high-noise and high-rate regimes. *IEEE Transactions on Information Theory* **63,** 1961–1970 (2017).

78. Haeupler, B. & Shahrasbi, A. *Synchronization strings: codes for insertions and deletions approaching the singleton bound* in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing* (2017), 33–46.

79. Guruswami, V. & Li, R. *Coding against deletions in oblivious and online models* in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms* (ed Czumaj, A.) (SIAM, 2018), 625–643.

80. Haeupler, B., Shahrasbi, A. & Sudan, M. *Synchronization Strings: List Decoding for Insertions and Deletions* in *ICALP* (eds Chatzigiannakis, I., Kaklamanis, C., Marx, D. & Sannella, D.) **107** (2018), 76:1–76:14.

81. Haeupler, B. & Shahrasbi, A. *Synchronization strings: Explicit constructions, local decoding, and applications* in *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing* (2018), 841–854.

82. Brakensiek, J., Guruswami, V. & Zbarsky, S. Efficient Low-Redundancy Codes for Correcting Multiple Deletions. *IEEE Trans. Inf. Theory* **64,** 3403–3410 (2018).

83. Cheng, K., Jin, Z., Li, X. & Wu, K. *Deterministic Document Exchange Protocols, and Almost Optimal Binary Codes for Edit Errors* in *FOCS* (ed Thorup, M.) (2018), 200–211.

84. Cheng, K., Haeupler, B., Li, X., Shahrasbi, A. & Wu, K. *Synchronization Strings: Highly Efficient Deterministic Constructions over Small Alphabets* in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms* (ed Chan, T. M.) (SIAM, 2019), 2185–2204.

85. Cheng, K., Jin, Z., Li, X. & Wu, K. *Block Edit Errors with Transpositions: Deterministic Document Exchange Protocols and Almost Optimal Binary Codes* in *ICALP* **132** (2019), 37:1–37:15.

86. Guruswami, V. & Li, R. Polynomial Time Decodable Codes for the Binary Deletion Channel. *IEEE Trans. Inf. Theory* **65,** 2171–2178 (2019).

87. Haeupler, B., Rubinstein, A. & Shahrasbi, A. *Near-linear time insertion-deletion codes and (1+ ε)-approximating edit distance via indexing* in *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing* (2019), 697–708.

88. Haeupler, B. *Optimal Document Exchange and New Codes for Insertions and Deletions* in *FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019* (ed Zuckerman, D.) (2019), 334–347.

89. Liu, S., Tjuawinata, I. & Xing, C. On List Decoding of Insertion and Deletion Errors. *CoRR* **abs/1906.09705** (2019).

90. Guruswami, V., Haeupler, B. & Shahrasbi, A. *Optimally resilient codes for list-decoding from insertions and deletions* in *STOC* (eds Makarychev, K., Makarychev, Y., Tulsiani, M., Kamath, G. & Chuzhoy, J.) (ACM, 2020), 524–537.

91. Cheng, K., Guruswami, V., Haeupler, B. & Li, X. *Efficient linear and affine codes for correcting insertions/deletions* in *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2021), 1–20.

92. Cheng, K. & Li, X. *Efficient document exchange and error correcting codes with asymmetric information* in *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2021), 2424–2443.

93. Sloane, N. On Single-Deletion-Correcting Codes. *arXiv: Combinatorics* (2002).

94. Mercier, H., Bhargava, V. K. & Tarokh, V. A survey of error-correcting codes for channels with symbol synchronization errors. *IEEE Communications Surveys and Tutorials* **12** (2010).

95. Mitzenmacher, M. *A Survey of Results for Deletion Channels and Related Synchronization Channels* in. **6** (July 2008), 1–3.

96. Haeupler, B. & Shahrasbi, A. *Synchronization Strings and Codes for Insertions and Deletions – a Survey* 2021. arXiv: 2101.00711 [cs.IT].

97. Ostrovsky, R. & Paskin-Cherniavsky, A. *Locally Decodable Codes for Edit Distance* in *Information Theoretic Security* (eds Lehmann, A. & Wolf, S.) (Springer International Publishing, Cham, 2015), 236–249.

98. Block, A. R., Blocki, J., Grigorescu, E., Kulkarni, S. & Zhu, M. *Locally Decodable/-Correctable Codes for Insertions and Deletions* in *FSTTCS* **182** (2020), 16:1–16:17.

99. Block, A. R. & Blocki, J. *Private and resource-bounded locally decodable codes for insertions and deletions* in *2021 IEEE International Symposium on Information Theory (ISIT)* (2021), 1841–1846.

100. Cheng, K., Li, X. & Zheng, Y. Locally decodable codes with randomized encoding. *arXiv preprint arXiv:2001.03692* (2020).

101. Blocki, J. *et al. Exponential Lower Bounds for Locally Decodable and Correctable Codes for Insertions and Deletions* in *2021 IEEE 62th Annual Symposium on Foundations of Computer Science (FOCS)* (2022), 739–750.

102. Kaufman, T. & Viderman, M. *Locally Testable vs. Locally Decodable Codes* in *APPROX-RANDOM* **6302** (Springer, 2010), 670–682.

103. Bhattacharyya, A., Gopi, S. & Tal, A. Lower bounds for 2-query LCCs over large alphabet. *arXiv preprint arXiv:1611.06980* (2016).

104. Ostrovsky, R., Pandey, O. & Sahai, A. *Private Locally Decodable Codes* in *ICALP* (2007), 387–398.

105. Yazdi, S. M. H. T., Gabrys, R. & Milenkovic, O. Portable and Error-Free DNA-Based Data Storage. *Scientific Reports* **7,** 2045–2322 (1 2017).

106. Blocki, J., Kulkarni, S. & Zhou, S. On Locally Decodable Codes in Resource Bounded Channels. *Leibniz International Proceedings in Informatics (LIPIcs)* **163** (eds Kalai, Y. T., Smith, A. D. & Wichs, D.) 16:1–16:23 (2020).

107. Lipton, R. J. *A new approach to information theory* in *Annual Symposium on Theoretical Aspects of Computer Science* (1994), 699–708.

108. Ding, Y., Gopalan, P. & Lipton, R. *Error Correction Against computationally bounded adversaries* Manuscript. 2004.

109. Micali, S., Peikert, C., Sudan, M. & Wilson, D. A. *Optimal Error Correction Against Computationally Bounded Noise* in *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings* (2005), 1–16.

110. Guruswami, V. & Smith, A. Optimal Rate Code Constructions for Computationally Simple Channels. *J. ACM* **63,** 35:1–35:37 (Sept. 2016).

111. Shaltiel, R. & Silbak, J. *Explicit List-Decodable Codes with Optimal Rate for Computationally Bounded Channels* in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM* (2016), 45:1–45:38.

112. Hemenway, B. & Ostrovsky, R. *Public-Key Locally-Decodable Codes* in *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Proceedings* (2008), 126–143.

113. Hemenway, B., Ostrovsky, R., Strauss, M. J. & Wootters, M. *Public Key Locally Decodable Codes with Short Keys* in *14th International Workshop, APPROX, and 15th International Workshop, RANDOM, Proceedings* (2011), 605–615.

114. Hemenway, B., Ostrovsky, R. & Wootters, M. Local correctability of expander codes. *Inf. Comput.* **243,** 178–190 (2015).

115. Blocki, J., Gandikota, V., Grigorescu, E. & Zhou, S. *Relaxed Locally Correctable Codes in Computationally Bounded Channels* in *ISIT* (IEEE, 2019), 2414–2418.

116. Smith, A. *Scrambling adversarial errors using few random bits, optimal information reconciliation, and better private codes* in *18th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007* (2007), 395–404.

117. Hirschberg, D. S. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM* **18,** 341–343 (1975).

118. Chakraborty, D., Das, D. & Koucký, M. *Approximate Online Pattern Matching in Sublinear Time* in *39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2019, December 11-13, 2019, Bombay, India* (eds Chattopadhyay, A. & Gastin, P.) **150** (Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019), 10:1–10:15.

119. Yamakami, T. *The 2CNF Boolean Formula Satisfiability Problem and the Linear Space Hypothesis* in *Proceedings of the 42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017)* **83** (2017), 62:1–62:14.

120. Tiskin, A. Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science* **1,** 571–603 (2008).

121. Andoni, A. & Krauthgamer, R. The Computational Hardness of Estimating Edit Distance. *SIAM Journal on Discrete Mathematics* **39,** 2398–2429 (2010).

122. Andoni, A., Jayram, T. & Patrascu, M. *Lower bounds for Edit Distance and Product Metrics via Poincare Type Inequalities* in *Proceedings of the twenty first annual ACM-SIAM symposium on Discrete algorithms* (2010), 184–192.

123. Kalyanasundaram, B. & Schintger, G. The probabilistic communication complexity of set intersection. *SIAM Journal on Discrete Mathematics* **5,** 545–557 (1992).

124. Razborov, A. A. *On the distributional complexity of disjointness* in *International Colloquium on Automata, Languages, and Programming* (1990), 249–253.

125. Kushilevitz, E. & Nisan, N. *Communication Complexity* (Cambridge Press, 1997).

126. Kaplan, E., Naor, M. & Reingold, O. Derandomized constructions of k-wise (almost) independent permutations. *Algorithmica* **55,** 113–133 (2009).

127. Cheng, K., Ishai, Y. & Li, X. *Near-Optimal Secret Sharing and Error Correcting Codes in* AC^0 in *Theory of Cryptography Conference* (2017), 424–458.

128. Ben-Sasson, E., Goldreich, O., Harsha, P., Sudan, M. & Vadhan, S. P. Robust PCPs of Proximity, Shorter PCPs, and Applications to Coding. *SIAM J. Comput.* **36.** A preliminary version appeared in the Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC), 889–974 (2006).

129. Gur, T., Ramnarayan, G. & Rothblum, R. D. *Relaxed Locally Correctable Codes* in *ITCS* (2018), 27:1–27:11.

130. Gur, T. & Lachish, O. On the Power of Relaxed Local Decoding Algorithms. *SIAM J. Comput.* **50,** 788–813 (2021).

Appendix A

Proofs of Results in Section 2.2

Proof of Lemma 2.2.1. Let $D_i = \mathsf{ED}(x^i, y[\alpha_i : \beta_i])$. Since we assume the alignment is optimal and $[\alpha_i : \beta_i]$ are disjoint and span the entire length of y, we know $\mathsf{ED}(x, y) = \sum_{i=1}^{b} D_i$.

For each $i \in [b]$, if $\varepsilon'|\alpha_i - \beta_i + 1| \leq |x^i| \leq 1/\varepsilon'|\alpha_i - \beta_i + 1|$, by the definition of (ε', Δ)-*approximately optimal candidate*, we know,

$$|\alpha_i - \alpha_i'| \leq \varepsilon' \frac{\Delta}{b} \tag{A.1}$$

and

$$|\beta_i - \beta_i'| \leq \varepsilon' \frac{\Delta}{b} + \varepsilon' \, \mathsf{ED}(x^i, y[\alpha_i : \beta_i]) \tag{A.2}$$

Also notice that we can transform $y[\alpha_i' : \beta_i']$ to $y[\alpha_i : \beta_i]$ with $|\alpha_i - \alpha_i'| + |\beta_i - \beta_i'|$ insertions and then transform $y[\alpha_i : \beta_i]$ to x^i with $\mathsf{ED}(x^i, y[\alpha_i : \beta_i])$ edit operations. We have

$$\mathsf{ED}(x^i, y[\alpha_i' : \beta_i']) \leq \mathsf{ED}(x^i, y[\alpha_i : \beta_i]) + |\alpha_i - \alpha_i'| + |\beta_i - \beta_i'| \tag{A.3}$$

Meanwhile, we can always transform $y[\alpha_i : \beta_i]$ to $y[\alpha_i' : \beta_i']$ with $|\alpha_i - \alpha_i'| + |\beta_i - \beta_i'|$ deletions and then transform $y[\alpha_i' : \beta_i']$ to x^i with $\mathsf{ED}(x^i, y[\alpha_i' : \beta_i'])$. We have

$$D_i' \geq D_i. \tag{A.4}$$

Combining A.1 A.2 A.3 and A.4, we have

$$D_i \le D'_i \le \mathsf{ED}(x^i, y[\alpha_i : \beta_i]) + 2|\alpha_i - \alpha'_i| + 2|\beta_i - \beta'_i| \le (1+2\varepsilon')D_i + 4\varepsilon' \frac{\Delta}{b}. \quad \text{(A.5)}$$

For those i such that $|x^i| > (1/\varepsilon')|\alpha_i - \beta_i + 1|$ or $|x^i| < \varepsilon'|\alpha_i - \beta_i + 1|$, to transform x^i to $y[\alpha_i : \beta_i]$, we need to insert (or delete) $||\alpha_i - \beta_i + 1| - |x^i||$ characters to make sure the length of x^i equals to the length of $y[\alpha_i : \beta_i]$. Thus, $D_i = \mathsf{ED}(x^i, y[\alpha_i : \beta_i])$ is at least $||\alpha_i - \beta_i| - |l_i - r_i||$. Since $D'_i = |\alpha_i - \beta_i| + |l_i - r_i|$, we have

$$\begin{aligned} D'_i \le & \frac{1+\varepsilon'}{1-\varepsilon'} D_i \\ \le & (1+3\varepsilon')D_i \quad \text{Since we set } \varepsilon' = \varepsilon/10 \le 1/10 \end{aligned} \quad \text{(A.6)}$$

Also notice that we can turn x^i into $y[\alpha_i : \beta_i]$with $|l_i - r_i|$ deletions and $|\alpha_i - \beta_i|$ insertions, we know $D'_i \ge D_i$. It gives us

$$D_i \le D'_i \le (1+3\varepsilon')D_i \quad \text{(A.7)}$$

Thus for each $i \in [b]$, by A.7 and A.5, we have

$$D_i \le D'_i \le (1+3\varepsilon')D_i + 4\varepsilon' \frac{\Delta}{N}. \quad \text{(A.8)}$$

Since we assume $\Delta \le (1+\varepsilon')\,\mathsf{ED}(x,y)$, we have $\varepsilon'\Delta \le 1.1\varepsilon'\,\mathsf{ED}(x,y)$, this gives us

$$\mathsf{ED}(x,y) \le \sum_{i=1}^{b} D'_i \le (1+3\varepsilon')\,\mathsf{ED}(x,y) + 4\varepsilon'\Delta \le (1+10\varepsilon')\,\mathsf{ED}(x,y) = (1+\varepsilon)\,\mathsf{ED}(x,y). \quad \text{(A.9)}$$

□

Proof of Lemma 2.2.2. Let $C^i_{\varepsilon,\Delta}$ be the output of $\mathsf{CandidateSet}(n, m, b, (l_i, r_i), \varepsilon, \Delta)$. For the starting point i', we only choose multiples of $\varepsilon\frac{\Delta}{b}$ from $[l_i - \Delta - \varepsilon\frac{\Delta}{b}, l_i + \Delta + \varepsilon\frac{\Delta}{b}]$. At most $O(\Delta/(\varepsilon\frac{\Delta}{b})) = O(b/\varepsilon)$ starting points will be chosen. For each starting point, we consider $O(\log_{1+\varepsilon} m) = O(\frac{\log m}{\varepsilon}) = O(\frac{\log n}{\varepsilon})$ ending point since we assume $\varepsilon m \le n \le \frac{1}{\varepsilon} m$. Thus, the size of set $C^i_{\varepsilon,\Delta}$ is at most $O(\frac{b \log n}{\varepsilon^2})$.

We now show there is an element in $C^i_{\varepsilon,\Delta} = \mathsf{CandidateSet}(n, m, b, (l_i, r_i), \varepsilon, \Delta)$ that is an (ε, Δ)-*approximately optimal candidate* of x^i if $\varepsilon|\alpha_i - \beta_i + 1| \le |x^i| \le 1/\varepsilon|\alpha_i - \beta_i + 1|$.

Since we assume $\Delta \ge \mathsf{ED}(x, y)$, we are guaranteed that $l_i - \Delta \le \alpha_i \le l_i + \Delta$. Thus, there is a multiple of $\lceil \varepsilon \frac{\Delta}{b} \rceil$, denoted by α', such that

$$l_i - \Delta - \varepsilon\frac{\Delta}{b} \le \alpha_i \le \alpha' \le \alpha_i + \varepsilon\frac{\Delta}{b} \le l_i + \Delta + \varepsilon\frac{\Delta}{b},$$

since we try every multiple of $\lceil \varepsilon \frac{\Delta}{b} \rceil$ between $l_i - \Delta - \varepsilon\frac{\Delta}{b}$ and $l_i + \Delta + \varepsilon\frac{\Delta}{b}$, one of them equals to α'.

For the ending point, we first consider the case when the length of $y[\alpha_i : \beta_i]$ is larger than the length of x^i, that is $\beta_i - \alpha_i + 1 \ge r_i - l_i + 1$. We know $\mathsf{ED}(x^i, y[\alpha_i : \beta_i]) \ge \beta_i - \alpha_i + 1 - |x^i|$. Let j be the largest element in $\{0, 1, \lceil 1 + \varepsilon \rceil, \lceil (1 + \varepsilon)^2 \rceil, \cdots, \lceil (1 + \varepsilon)^{\log_{1+\varepsilon}(m)} \rceil\}$ such that $\alpha' + |x^i| - 1 + j \le \beta_i$. We set $\beta' = \alpha' + |x^i| - 1 + j$. Since $j \ge (\beta_i - (\alpha' + |x^i| - 1))/(1 + \varepsilon)$, we have

$$\begin{aligned}\beta' \ge& \alpha' + |x^i| - 1 + (\beta_i - (\alpha' + |x^i| - 1))/(1 + \varepsilon)\\ \ge& \frac{\beta_i}{1+\varepsilon} + \frac{\varepsilon}{1+\varepsilon}(\alpha' + |x^i| - 1)\\ \ge& \beta_i - \frac{\varepsilon}{1+\varepsilon}(\beta_i - \alpha' + 1 - |x^i|)\\ \ge& \beta_i - \varepsilon\,\mathsf{ED}(x^i, y[\alpha_i : \beta_i])\end{aligned} \tag{A.10}$$

The last inequality is because $\mathsf{ED}(x^i, y[\alpha_i : \beta_i]) \ge \beta_i - \alpha_i + 1 - |x^i| \ge \beta_i - \alpha' + 1 - |x^i|$ and $\varepsilon \ge \frac{\varepsilon}{1+\varepsilon}$. Thus, $(\alpha', \beta') \in C^i_{\varepsilon,\Delta}$ is an (ε, Δ)-*approximately optimal candidate* of x^i.

For the case when $\beta_i - \alpha_i + 1 < |x^i|$. Similarly, we know $\mathsf{ED}(x^i, y[\alpha_i : \beta_i]) \ge |x^i| - (\beta_i - \alpha_i + 1)$.We pick j to be the smallest element in $\{0, 1, \lceil 1 + \varepsilon \rceil, \lceil (1 + \varepsilon)^2 \rceil, \cdots, \lceil (1 + \varepsilon)^{\log_{1+\varepsilon}(m)} \rceil\}$ such that $\alpha' + |x^i| - 1 - j \le \beta_i$. We know $j \le (1 + \varepsilon)(\alpha' + |x^i| - 1 - \beta_i)$.

We set $\beta' = \alpha' + |x^i| - j$. Then

$$\begin{aligned}
\beta' \geq & \alpha' + |x^i| - 1 - (1+\varepsilon)(\alpha' + |x^i| - 1 - \beta_i) \\
\geq & \beta_i - \varepsilon(\alpha' - \beta_i + |x^i| - 1) \\
\geq & \beta_i - \varepsilon(\alpha_i + \varepsilon\frac{\Delta}{N} - \beta_i + |x^i| - 1) \\
\geq & \beta_i - \varepsilon \, \mathsf{ED}(x^i, y[\alpha_i : \beta_i]) - \varepsilon^2 \frac{\Delta}{b} \\
\geq & \beta_i - \varepsilon \, \mathsf{ED}(x^i, y[\alpha_i : \beta_i]) - \varepsilon \frac{\Delta}{b}
\end{aligned} \tag{A.11}$$

Thus, $(\alpha', \beta') \in C^i_{\varepsilon,\Delta}$ is an (ε, Δ)-*approximately optimal candidate* of x^i.

□

Proof of Lemma 2.2.3. We start by explaining the dynamic programming. Let f be a function such that $f(i) \in C^i_{\varepsilon',\Delta} \cup \{\emptyset\}$. We say an interval x^i is matched if $f(i) \in C^i_{\varepsilon',\Delta}$ and it is unmatched if $f(i) = \emptyset$. Let S^f_1 be the set of indices of matched blocks under function f and $S^f_2 = [b] \setminus S^f_1$ be the set of indices of unmatched blocks. We let $f(i) = (\alpha^f_i, \beta^f_i)$ for each $i \in S^f_1$. We also require that, for any $i, j \in S^f_1$ with $i < j$, (α^f_i, β^f_i) and (α^f_j, β^f_j) are disjoint and $\beta^f_i < \alpha^f_j$. Let u_f be the number of unmatched characters under f in x and y. That is, u_f equals to the number of indices in $[n]$ that is not in any matched block plus the number of indices in $[m]$ that is not in $f(i)$ for any $i \in S^f_1$. Then we define the edit distance under match f by

$$\mathsf{ED}_f := \sum_{i \in S^f_1} \mathsf{ED}(x^i, y[\alpha^f_i : \beta^f_i]) + u_f.$$

Since we can always transform x to y by deleting (inserting) every unmatched characters in x (y), and transforming each matched block x^i into $y[\alpha^f_i : \beta^f_i]$ with $\mathsf{ED}(x^i, y[\alpha^f_i : \beta^f_i])$ edit operations. We know $\mathsf{ED}_f \geq \mathsf{ED}(x, y)$

Let F be the set of all matchings. Also, given $i \in [b]$ and $\alpha \in [m]$, we let $F^{i,\alpha}$ be the set of matching such that $f(i')$ is within $(1, \alpha)$ for all $i' \leq i$. Similarly, for each $f \in F^{i,\alpha}$, let $u^{i,\alpha}_f$ be the number of unmatched characters in $x[1, r_i]$ and $y[1 : \alpha]$ under f. We can also define $\mathsf{ED}^{i,\alpha}_f = \sum_{i \in S^f_1} \mathsf{ED}(x^i, y[\alpha^f_i : \beta^f_i]) + u^{i,\alpha}_f$. For simplicity, let C^i be the set of starting points of all intervals in $C_{\varepsilon,\Delta}^{i+1}$. We now show that in Algorithm 3, for each $i \in [b-1]$ and $\alpha \in C^{i+1}$, we have

$$A(i, \alpha - 1) = \min_{f \in F^{i,\alpha-1}} \mathsf{ED}_f^{i,\alpha-1}.$$

We can proof this by induction on i. For the base case $i = 1$, we fix an $\alpha \in C^2$. For each $f \in F^{1,\alpha-1}$, if $f(1) = \emptyset$, then every character in x^1 and $y[1 : \alpha - 1]$ are unmatched. In this case, $\mathsf{ED}_f^{1,\alpha-1} = |x^1| + \alpha - 1 = A(0, \alpha' - 1) + \alpha - \alpha' + |x^1|$ for every $\alpha' \in C^1$ such that $\alpha' \leq \alpha$. When $f(1) \neq \emptyset$, we assume $f(1) = (\alpha_1^f, \beta_1^f)$, then $\mathsf{ED}_f^{i,\alpha-1} = \alpha_1^f - 1 + M(1, (\alpha_i^f, \beta_i^f)) + \alpha - \beta = A(0, \alpha^f) + M(1, (\alpha_i^f, \beta_i^f)) + \alpha - \beta$. By the updating rule of $A(1, \alpha - 1)$ at line 6, we know $A(1, \alpha - 1) = \min_{f \in F^{1,\alpha-1}} \mathsf{ED}_f^{1,\alpha-1}$ for every $\alpha \in C^2$.

Now assume $A(t-1, \alpha - 1) = \min_{f \in F^{t-1,\alpha-1}} \mathsf{ED}_f^{t-1,\alpha-1}$ for any $\alpha \in C^t$ for $1 < t \leq b - 1$. Fix an $\alpha_0 \in C^{t+1}$, we show $A(t, \alpha_0 - 1) = \min_{f \in F^{t,\alpha_0-1}} \mathsf{ED}_f^{t,\alpha_0-1}$. For each matching f, if $f(t) = \emptyset$, we know

$$\mathsf{ED}_f^{t,\alpha_0-1} = \mathsf{ED}_f^{t-1,\alpha_0-1} + |x^t| \geq \min_{\alpha' \in C^t, \alpha' \leq \alpha} A(t-1, \alpha' - 1) + \alpha_0 - \alpha' \geq A(t, \alpha_0 - 1).$$

When $f(t) \neq \emptyset$, we assume $f(t) = (\alpha_t^f, \beta_t^f)$. Then

$$\begin{aligned} \mathsf{ED}_f^{t,\alpha_0-1} &= \mathsf{ED}_f^{t-1,\alpha_t^f-1} + M(t, (\alpha_t^f, \beta_t^f)) + \alpha_0 - \beta_t^f - 1 \\ &\geq A(t-1, \alpha_t^f - 1) + M(t, (\alpha_t^f, \beta_t^f)) + \alpha_0 - \beta_t^f - 1 \\ &\geq A(t, \alpha_0 - 1) \end{aligned}$$

Meanwhile, $A(t, \alpha_0) \geq \min_{f \in F^{t,\alpha_0-1}} \mathsf{ED}_f^{t,\alpha_0-1}$ since $A(t, \alpha_0) = \mathsf{ED}_f^{t,\alpha_0-1}$ for some $f \in F^{t,\alpha_0-1}$ by the updating rule at line 6. Thus, we have proved $A(t, \alpha_0 - 1) = \min_{f \in F^{t,\alpha_0-1}} \mathsf{ED}_f^{t,\alpha_0-1}$. Now, assume we have computed $A(b-1, \alpha)$ for every $\alpha \in C^b$. Let f_0 be the optimal matching such that $\mathsf{ED}_{f_0}(x, y) = \min_{f \in F} \mathsf{ED}_f(x, y)$. If $f_0(b) = \emptyset$,

$$\mathsf{ED}_{f_0}(x, y) = \min_{\alpha' \in C^b} A(b-1, \alpha' - 1) + |m - \alpha'| + |x^b|$$

Otherwise, let $f_0(b) = (\alpha_b^{f_0}, \beta_b^{f_0})$

$$\mathsf{ED}_{f_0}(x,y) = \min_{(\alpha',\beta')\in C^b_{\varepsilon,\Delta}} A(b-1,\alpha'-1) + M(b,(\alpha_b^{f_0},\beta_b^{f_0})) + m - \beta'$$

By the optimality of f_0, we know Algorithm 3 is $d = \mathsf{ED}_{f_0}(x,y)$. Now, if we fix an optimal alignment such that $x[l_i, r_i]$ is matched to block $y[\alpha_i : \beta_i]$ and $[\alpha_i, \beta_i]$ are disjoint and span the entire length of y. Let f_1 be a matching such that, for each $i \in [b]$, if $\varepsilon'|\alpha_i - \beta_i| \leq |l_i - r_i| \leq 1/\varepsilon'|\alpha_i - \beta_i|$, $f(i)$ is an (ε', Δ)-*approximately optimal candidate*. Otherwise, $f(i) = \emptyset$. By lemma 2.2.1 and Lemma 2.2.2, such a matching f_1 exists and $\mathsf{ED}_f \leq (1+\varepsilon)\,\mathsf{ED}(x,y)$. Thus,

$$\mathsf{ED}(x,y) \leq \mathsf{ED}_{f_0} \leq \mathsf{ED}_{f_1} \leq (1+\varepsilon)\,\mathsf{ED}(x,y)$$

This proves the correctness of Algorithm 3.

Now we compute the tme complexity. By the proof of Lemma 2.2.2, $|C^i| = O(\frac{b}{\varepsilon})$ for $i \in [b]$. The size of matrix A is $O(\frac{b^2}{\varepsilon})$ where the rows of A are indexed by i from 0 to $b-1$ and for the i-th row, the columns are indexed by the elements in set C^{i+1}. We can divide the dynamic programming into roughly b steps and for the i-th step, we compute the row indexed by i. Assume we have already computed the row indexed by $i-1$ of A. We first set

$$A(i,\alpha-1) = \min_{\alpha'\in C^i, \alpha'\leq\alpha} A(i-1,\alpha'-1) + \alpha - \alpha' + |x^i|$$

for all $\alpha \in C^i$. This takes $O(|C^i||C^{i+1}|) = O(\frac{b^2}{\log^2 n})$ time. Then, we query each elements in the i-th row of M. Say we queried $M(i,(\alpha',\beta'))$, we update all $A(i,\alpha-1)$ such that $\alpha - 1 \geq \beta'$ by

$$A(i,\alpha-1) = \min\{A(i,\alpha-1), A(i-1,\alpha'-1) + M(i,(\alpha',\beta')) + \alpha - 1 - \beta'\}.$$

This takes $O(|C^i_{\varepsilon',\Delta}||C^{i+1}|) = O(\frac{b^2}{\varepsilon^3}\log n)$ time. So the i-th step takes $O(\frac{b^2}{\varepsilon^3}\log n)$ time. SInce there are b steps, the time complexity is bound by $O(\frac{b^3}{\varepsilon^3}\log n)$.

For the space complexity, notice when updating $A(i, \alpha)$, we only need the information of $A(i-1, \alpha'-1)$ for every $\alpha' \in C^i$. Thus, we can release the space used to store $A(i-2, \alpha''-1)$ for every $\alpha'' \in C^{i-1}$. And for line 7, we only need the information of $A(i-1, \alpha-1)$ for every $\alpha \in C^i$. From Algorithm 2, we know that for each i, we pick at most b/ε points as the starting point of the candidate intervals. The size of C^i is at most b/ε. Since each element in A is a number at most n, it can be stored with $O(\log n)$ bits of space. Thus, the space required is $O(\frac{b}{\varepsilon} \log n)$.

If we replace $M(i, (\alpha, \beta))$ with a $(1+\gamma)$ approximation of $\mathsf{ED}(x^i, y[\alpha : \beta])$. Each $M(i, (\alpha, \beta))$ will add at most an $\gamma \, \mathsf{ED}(x^i, y[\alpha : \beta])$ additive error. The amount of error added is bounded by $\gamma \, \mathsf{ED}(x, y)$. Thus, $\mathsf{EditDP}(n, m, b, \varepsilon', \Delta, M)$ outputs a $(1+\varepsilon)(1+\gamma)$-approximation of $\mathsf{ED}(x, y)$. The time and space complexity is not affected. $\square$

Appendix B

A Simple Lower Bound for Edit Distance in the Standard Streaming Model

Theorem B.1. *There exists a constant $\varepsilon > 0$ such that for strings $x, y \in \{0,1\}^n$, any deterministic R pass streaming algorithm achieving an εn additive approximation of $\mathsf{ED}(x, y)$ needs $\Omega(n/R)$ space.*

Proof. Consider an asymptotically good insertion-deletion code $C \subseteq \{0,1\}^n$ over a binary alphabet (See [75] for example). Assume C has rate α and distance β. Both α and β are some constants larger than 0, and we have $|C| = 2^{\alpha n}$. Also, for any $x, y \in C$ with $x \neq y$, we have $\mathsf{ED}(x, y) \geq \beta n$. Let $\varepsilon = \beta/2$ and consider the two party communication problem where player 1 holds $x \in C$ and player 2 holds $y \in C$. The goal is to decide whether $x = y$. Any deterministic protocol has communication complexity at least $\log |C| = \Omega(n)$. Note that any algorithm that approximates $\mathsf{ED}(x, y)$ within an εn additive error can decide whether $x = y$. Thus the theorem follows. $\square$

We note that the same bound holds for Hamming distance by the same argument.