Finding Longest Increasing and Common Subsequences in Streaming Data*

David Liben-Nowell
Department of Computer Science
Carleton College
dlibenno@carleton.edu

Erik Vee
IBM Almaden Research Center
vee@almaden.ibm.com

An Zhu
Google, Inc.
anzhu@google.com

Abstract

We present algorithms and lower bounds for the Longest Increasing Subsequence (LIS) and Longest Common Subsequence (LCS) problems in the data-streaming model. To decide if the LIS of a given stream of elements drawn from an alphabet Σ has length at least k, we discuss a one-pass algorithm using $O(k \log |\Sigma|)$ space, with update time either $O(\log k)$ or $O(\log \log |\Sigma|)$; for $|\Sigma| = O(1)$, we can achieve $O(\log k)$ space and constant-time updates. We also prove a lower bound of $\Omega(k)$ on the space requirement for this problem for general alphabets Σ, even when the input stream is a permutation of Σ. For finding the actual LIS, we give a $\lceil \log(1 + 1/\varepsilon) \rceil$-pass algorithm using $O(k^{1+\varepsilon} \log |\Sigma|)$ space, for any $\varepsilon > 0$. For LCS, there is a trivial $\Theta(1)$-approximate $O(\log n)$-space streaming algorithm when $|\Sigma| = O(1)$. For general alphabets Σ, the problem is much harder. We prove several lower bounds on the LCS problem, of which the strongest is the following: it is necessary to use $\Omega(n/\rho^2)$ space to approximate the LCS of two n-element streams to within a factor of ρ, even if the streams are permutations of each other.

1 Introduction

Longest increasing and common subsequences. Let $\mathcal{S} = \langle x_1, x_2, \ldots, x_n \rangle$ be a sequence of integers. A *subsequence* of $\mathcal{S}$ is a sequence $\langle x_{i_1}, x_{i_2}, \ldots, x_{i_k} \rangle$ with $i_1 < i_2 < \cdots < i_k$. A subsequence is *increasing* if $x_{i_1} \leq x_{i_2} \leq \cdots \leq x_{i_k}$. We consider two problems related to subsequences:

- *longest increasing subsequence* (LIS): given a sequence $\mathcal{S}$, find a maximum-length increasing subsequence of $\mathcal{S}$ (or find the length of such a sequence); and
- *longest common subsequence* (LCS): given two sequences $\mathcal{S}$ and $\mathcal{T}$, find a maximum-length sequence x that is a subsequence of both $\mathcal{S}$ and $\mathcal{T}$ (or find the length of x).

*Appears in *Journal of Combinatorial Optimization*, Volume 11, Number 2, March 2006, pp. 155–175. A preliminary version of this paper appears in the *Proceedings of the 11th International Computing and Combinatorics Conference (COCOON'05)*, August 2005, pp. 263–272. This document was last updated on 5 May 2006. Comments are welcome.

Both LIS and LCS are fundamental combinatorial questions that have been well studied within the computer-science literature (e.g., [4, 8, 18, 23, 24, 31], among many others).

Among a large number of important applications of both of these problems, we highlight a few that arise in computational biology. The BLAST (Basic Local Alignment Search Tool) database [3] supports queries of the following form: for a sequence σ of amino acids, for example, what segments of known proteins have high local similarity to σ? Zhang [34] has proposed filtering the results of a BLAST query with an approach that uses an LIS algorithm as a black box to assemble BLAST information about local similarity into a coherent picture of global similarity. An LIS step is also part of the MUMmer system for aligning entire genomes [12], and a straightforward LCS computation gives the value of the optimal alignment of two sequences of DNA [30].

The data-streaming model. In the past few years, as we have witnessed the proliferation of massive data sets as diverse as fully sequenced genomes and the link structure of the World Wide Web, traditional notions of algorithmic efficiency have begun to appear inadequate. A polynomial-time algorithm—normally seen as the theoretical holy grail for a problem—may simply not be fast enough when it is run on an input as big as the multi-billion base pairs of the human genome.

Researchers in theoretical computer science has thus begun to explore new models of computation, with new notions of efficiency, that more realistically capture when an algorithm is "fast enough." The *data-streaming model* [22] is one such well-studied model. In this model, an algorithm must make a small number of passes over the input data, processing each input element as it goes by. Once the algorithm has seen an element, it is gone forever; thus we must compute and store a small amount of useful information about the previously read input. We are interested in algorithms that use a sublinear amount of additional space. (With a linear amount of space, a streaming algorithm can simply store the entire input and then run a traditional algorithm.) We typically aim for a polylogarithmic amount of space and a polylogarithmic amount of processing time for each element of the input. Ideal data-streaming algorithms make only a single pass over the data, but we are also interested in *multipass* streaming algorithms, in which the algorithm can make a small number (typically constant) of passes over the input data.

LIS/LCS in the data-streaming model. In this paper, we study the difficulty of the LIS and LCS problems in the data-streaming model. We are motivated in our exploration by the fact that LIS and LCS are fundamental combinatorial questions arising naturally in the streaming context that are essentially different from other problems previously studied in this model. We believe that a solid characterization of the tractability of basic questions like LIS and LCS will lead to a greater understanding of the power and limitations of the data-streaming model.

Applications for LIS and LCS also arise in many natural settings. For example, the optimal-alignment problem for two DNA sequences requires the computation of the LCS of two genomes, strings of length potentially far in excess of the size of main memory. Sequentially fetching blocks of data from disk is significantly faster than random block fetches; thus sequentially streaming the data from disk, using a small number of passes over the data, is highly desirable. (This approach has been considered in the external-memory community for other problems as well [7, 15].)

Another potential application for LIS and LCS in the streaming model is that, in certain real-life settings, high-speed data can pass by a bounded-memory device—e.g., a stream of packets passing a router—and we may wish to perform some sort of computation on the stream. The question of "what is different about this data stream now, as compared to yesterday?" has been studied from

the perspective of large changes in the frequencies of particular elements in the stream [11]; LCS looks at the same question from the perspective of changes in the order of elements in the stream. In a related application, Banerjee and Ghosh [5] have explored the use of LCS as a mechanism for clustering the users of a website on the basis of their "clickstreams" through the site.

One notable difference between LIS (and, similarly, LCS) and problems that have been previously considered in the data-streaming model is that the LIS of a stream is an essentially global *order-based* property. Many of the problems that have been considered in the streaming model—for example, finding the most frequently occurring items in a stream [9, 13], clustering streaming data [21], finding order statistics for a given stream [2, 27], or calculating the distance between two vectors presented as a stream of ordered pairs (x_i, i) of values and indices [2, 6, 16, 17, 25, 29]—are entirely independent of the order in which the elements are presented in $\mathcal{S}$; permuting the items in the stream does not affect the correct answers to these questions. Two exceptions are (1) counting *inversions* in a stream [1]—i.e., the number of pairs of indices (i, j) such that $i < j$ but $x_i > x_j$, and (2) computing a stream's *histogram* [19, 20]—i.e., a compact approximation of the stream by a piecewise constant function. However, there are some significant distinctions between these problems and LIS. Counting inversions is a much more local problem than LIS, in the sense that an inversion is a relationship between exactly two items in the stream, whereas an increasing subsequence of length ℓ is a relationship among ℓ items. Histograms are much more robust than LIS to small changes in the data: if we consider an LIS that consists primarily of the same repeated value, and we perturb the input so that many occurrences of this value are slightly smaller, then the LIS radically changes. Thus algorithms that store some sort of approximation of the stream's previous elements face significant obstacles in solving the LIS and LCS problems. While these differences do not preclude the existence of efficient streaming algorithms for LIS or LCS, they do suggest some of the difficulties.

Our results. In this paper, we give a full characterization (up to logarithmic factors) of LCS in the data-streaming model, even in the context of approximation. We also fully characterize (again up to logarithmic factors) the exact version of LIS, leaving approximations for future work. We first present positive results on (1) computing the length of the LIS of a given input stream, and (2) outputting a maximum-length increasing sequence itself. Fredman's algorithm [18], which was originally developed outside the context of the data-streaming model but which can be naturally interpreted as a streaming algorithm, yields a one-pass streaming algorithm that uses $O(k \log |\Sigma|)$ space with update time $O(\log k)$ to compute the length of the LIS for a given input stream, where the n elements of the stream are drawn from the (ordered) alphabet $\Sigma = \{1, \ldots, |\Sigma|\}$, and k is the length of the LIS. This algorithm can also achieve an update time of $O(\log \log |\Sigma|)$ if it is implemented using van Emde Boas queues or y-fast trees [32, 33]. For the problem of *returning* the length-k LIS of a given stream, Fredman's approach gives a one-pass streaming algorithm that uses $O(k^2 \log |\Sigma|)$ space. We reduce the space requirement to $O(k^{1+\varepsilon} \log |\Sigma|)$ by using $\lceil \log(1 + 1/\varepsilon) \rceil$ passes over the data, for any $\varepsilon > 0$. This space usage is nearly optimal, because merely storing the LIS itself requires $\Omega(k \log |\Sigma|)$ space when $|\Sigma| = \Omega(n)$. When the elements of the stream are drawn from a small alphabet, we can achieve $O(|\Sigma| \log k)$ space and $O(\log |\Sigma|)$ update time—i.e., logarithmic space and constant-time updates if $|\Sigma| = O(1)$—for computing the length of the LIS. For finding the LIS itself, we achieve the same bounds using $O(|\Sigma|^2 \log k)$ space.

We also present (the first, to the best of our knowledge) lower bounds on LIS/LCS in the streaming model in Section 4. (In the comparison model, Fredman has proven lower bounds on

computing LIS, via a reduction from sorting [18].) As with many lower bounds in the streaming model, our results are based upon the well-observed connection between the space required by a streaming algorithm to solve a problem and the problem's communication complexity. Specifically, a space-efficient streaming algorithm $\mathcal{A}$ to solve a problem gives rise to a solution to the corresponding two-party problem with low communication complexity: one party runs $\mathcal{A}$ on the first part of the input, transmits the small state of the algorithm to the other party, who then continues to run $\mathcal{A}$ on the remainder of the input. We prove a lower bound of $\Omega(k)$ space for computing the LIS of a stream whenever $n = \Omega(k^2)$ by giving a reduction from the Set-Disjointness problem, which is known to have high communication complexity.

In Section 5, we turn to the LCS problem. For solving LCS in the data-streaming model, we note a simple LIS-based algorithm requiring $O(n \log |\Sigma|)$ space to compute the LCS of two n-element sequences presented as streams. If we want to compute the LCS of one n-element *reference permutation* against any number of test permutations, we can achieve the same space bound, independent of the number of test permutations. For small alphabets, we can also approximate LCS in small space: we can achieve a simple $|\Sigma|$-approximation using $O(|\Sigma| \log n)$ space and $O(1)$ update time. Our main results on LCS, however, are lower bounds. We prove that if the two streams are general sequences—that is, not necessarily permutations of each other—then we need $\Omega(n)$ space to ρ-approximate the LCS of two streams of length n to within any factor ρ. (Note that the lower bound is independent of the desired approximation ratio.) Finally, we prove our strongest lower bound: if the given streams are n-element permutations, then $\Omega(n/\rho^2)$ space is required to ρ-approximate the LCS.

2 Notation

Throughout this paper, we will use n to denote the length of the input streams, $\Sigma = \{1, 2, \ldots, |\Sigma|\}$ to denote the (integral) alphabet from which elements of the stream are drawn, and k to denote the length of the longest increasing or common subsequence of the input stream(s). We will write sequences and subsequences in angle brackets (e.g., $\mathcal{S} = \langle x_1, \ldots, x_n \rangle$).

For any sequence σ, we let σ_i denote the ith element in σ. Let $|\sigma|$ denote the length of σ. Write $\mathsf{last}(\sigma) := \sigma_{|\sigma|}$. Define $\mathsf{all\text{-}but\text{-}last}(\sigma) = \langle \sigma_1, \sigma_2, \ldots, \sigma_{|\sigma|-1} \rangle$ as the $(|\sigma| - 1)$-element sequence resulting from removing from σ its last element $\mathsf{last}(\sigma) = \sigma_{|\sigma|}$. Let $\langle\rangle$ denote the empty sequence. For a single element x_i, let $\langle x_i \rangle$ denote the one-element sequence that contains the lone element x_i. For two sequences $\sigma = \langle \sigma_1, \ldots, \sigma_{|\sigma|} \rangle$ and $\sigma' = \langle \sigma'_1, \ldots, \sigma'_{|\sigma'|} \rangle$, we denote the concatenation of the streams by $\mathsf{concat}(\sigma, \sigma') = \langle \sigma_1, \ldots, \sigma_{|\sigma|}, \sigma'_1, \ldots, \sigma'_{|\sigma'|} \rangle$.

Finally, for any value $x \geq 0$, we say that $\widetilde{x}$ is a ρ-approximation of x if $x \leq \widetilde{x} \leq \rho x$.

3 Streaming Algorithms for Longest Increasing Subsequence

We begin by presenting positive results on the LIS problem, both for computing the length of an LIS and for actually producing an LIS itself. A small-space algorithm to calculate the length of the LIS was given by Fredman [18]—and later rediscovered by Bespamyatnikh and Segal [8]—in a context other than the data-streaming model. When it is viewed in the data-streaming context, this algorithm yields a one-pass algorithm to compute the length of the LIS of a given stream using $O(k \log |\Sigma|)$ space and update time of $O(\log k)$ or $O(\log \log |\Sigma|)$, where k is the length of the stream's LIS. The algorithm also naturally gives a $O(k^2 \log |\Sigma|)$-space algorithm to find a longest increasing

Fredman($\mathcal{S}$): *// compute the length of the LIS of the stream* $\mathcal{S} = \langle x_1, \ldots, x_n \rangle$.

1. Initialize $k' := 0$, $A[0] := -\infty$, and $A[1] := \infty$.

   // k' *is the length of the LIS in the stream seen so far.*
   // $A[i]$ *stores the smallest possible last element in a length-*i *increasing subsequence in the stream so far.*

2. While there are elements left in the stream $\mathcal{S}$:
   (a) read the next element x_i from $\mathcal{S}$.
   (b) find the ℓ such that $A[\ell] \leq x_i < A[\ell+1]$. *// using binary search or van Emde Boas queues.*
   (c) set $A[\ell+1] := x_i$.
   (d) if $\ell + 1 > k'$, then increment k' and initialize $A[k'+1] := \infty$.

3. Return k'.

Figure 1: Fredman's algorithm to compute the length of the LIS in stream $\mathcal{S}$.

subsequence itself. Here we briefly describe Fredman's algorithm, then we give a modification of this algorithm to handle the case of $|\Sigma| = O(1)$, and finally we extend this approach to a more space-efficient multipass streaming algorithm to compute an LIS itself.

3.1 Computing the Length of an LIS

Let $\mathcal{S} = \langle x_1, x_2, \ldots, x_n \rangle$ be a stream of data, and consider a length-ℓ increasing subsequence $\sigma = \langle x_{i_1}, x_{i_2}, \ldots, x_{i_\ell} \rangle$ of $\mathcal{S}$. We say that σ is (ℓ, j)-*minimal* if $\mathsf{last}(\sigma)$ is minimized over all length-ℓ increasing subsequences of the substream $\langle x_1, x_2, \ldots, x_j \rangle$. We say that such a sequence σ is an (ℓ, j)-*minimal increasing sequence*, or simply an (ℓ, j)-*MIS*.

The streaming algorithm of Fredman [18] for computing the LIS's length is based on storing the last element of an (ℓ, j)-MIS in an array cell $A[\ell]$ for all $\ell \in \{1, \ldots, k'\}$ as we scan the stream, where k' is the length of the LIS in the stream seen so far. Note that, by definition, the elements of A must be sorted in increasing order: if $\sigma = \langle \sigma_1, \ldots, \sigma_{\ell+1} \rangle$ is an $(\ell+1, j)$-MIS, then it necessarily contains an increasing length-ℓ subsequence $\langle \sigma_1, \ldots, \sigma_\ell \rangle$ where $\sigma_\ell \leq \sigma_{\ell+1}$. We update $A[\ell+1] := x_j$ when the next element x_j in the stream falls between $A[\ell]$ and $A[\ell+1]$, the currently stored values for the last element of the (ℓ, j)-MIS and $(\ell+1, j)$-MIS. The fact that the elements of A appear in sorted order means that the update step only requires finding the predecessor $A[\ell]$ of the stream element x_j in A. Pseudocode for the algorithm is shown in Figure 1.

Lemma 3.1. *Consider any stream* $\mathcal{S} = \langle x_1, \ldots, x_n \rangle$ *and any index* $i \in \{1, \ldots, n\}$. *After* i *iterations of the while loop in* Fredman($\mathcal{S}$), *we have that* k' *is the length of the LIS of* $\langle x_1, \ldots, x_i \rangle$ *and*

$$A[\ell] = \begin{cases} -\infty & \text{if } \ell = 0, \\ \mathsf{last}(\rho) \text{ for an } (\ell, i)\text{-MIS } \rho & \text{if } 1 \leq \ell \leq k', \\ \infty & \text{if } \ell = k'+1, \text{ and} \\ \text{uninitialized} & \text{otherwise.} \end{cases}$$

Proof. We first strengthen the stated property by adding the previously mentioned sortedness condition to the induction hypothesis: for all j and for all $j' > j$ such that $A[j]$ and $A[j']$ are both initialized, we have that $A[j] \leq A[j']$.

The proof follows by induction, in a relatively straightforward manner. For the base case $i = 0$, the property is vacuously true. For the inductive case, assume that the desired properties were maintained after we read the element x_{i-1} from the stream. Now consider the moment at which we read the next element x_i from the stream. Let ℓ be such that $A[\ell] \leq x_i < A[\ell + 1]$, as in the algorithm. It is clear that only MIS's of length $\ell + 1$ or more might have a new smallest last element, because every shorter MIS already consists entirely of elements that are smaller than x_i. In other words, the newly read element x_i can only affect values in A with indices $\ell + 1$ or higher.

On the other hand, note that x_i can only extend an existing increasing subsequence σ into a longer increasing subsequence if σ ends with some element $\sigma_{|\sigma|} \leq x_i$. For all such subsequences, we have $\sigma_{|\sigma|} \leq x_i < A[\ell + 1]$ by the definition of ℓ. Hence by the induction hypothesis, the sequence σ has length ℓ or shorter. But then the sequence $\sigma' = \mathsf{concat}(\sigma, \langle x_i \rangle)$ is of length at most $\ell + 1$. Thus the element x_i can only affect values in A with indices $\ell + 1$ or lower.

Indeed, we now have a new subsequence σ' of length $\ell + 1$ with x_i as the last element, by extending the (ℓ, i)-minimal increasing subsequence by adding the last element $A[\ell]$. Thus it is necessary and sufficient to update $A[\ell + 1]$. It is immediately clear that the requirement on k' is maintained, and it also clear that the new $A[j]$'s respect the sortedness constraint. □

Theorem 3.2. *We can decide whether the LIS of a stream of integers drawn from alphabet Σ has length at least some given number k, or compute the length k of the LIS of the given stream, with a one-pass streaming algorithm that uses $O(k \log |\Sigma|)$ space and has update time $O(\log k)$ or $O(\log \log |\Sigma|)$ per element.*

Proof. By Lemma 3.1, the length of the LIS is correctly computed by Fredman. Clearly, the decision problem can also be solved with a minor change to the output of this algorithm.

For the space bound, observe that we store k values from the stream—that is, k values in the range $\{1, \ldots, |\Sigma|\}$—in the array A, requiring $O(\log |\Sigma|)$ bits each. The only superconstant-time step in the update operation is to find the ℓ such that $A[\ell] \leq x_i < A[\ell + 1]$. This can be done in $O(\log k)$ time by explicitly storing the array A and using binary search; alternatively, we can use van Emde Boas queues [32] or y-fast trees [33] to support updates in $O(\log \log |\Sigma|)$ time. □

When the alphabet is of small size, we can modify this algorithm to be much more efficient in terms of both its update time and its space requirements. Instead of storing k different values drawn from the stream in the array A, we instead maintain the length of the longest increasing subsequence that has a particular element of the alphabet as its last element, for every alphabet character, as we read the stream. Specifically, consider an array $B[1 \ldots |\Sigma|]$ such that $B[a]$ denotes the length of the longest increasing subsequence $\sigma^{(a)}$ ending with $a \in \Sigma$. When a new element x_i of the stream arrives, we simply reset $B[x_i] := 1 + \max\{B[1], \ldots, B[x_i - 1], B[x_i]\}$. The length of the LIS is then $\max\{B[1], \ldots, B[|\Sigma|]\}$. Hence we can compute the length of the LIS exactly using $O(|\Sigma| \log k)$ space, with $O(|\Sigma|)$ update time.

The update time can be improved to $O(\log |\Sigma|)$ by placing a complete binary tree on top of the array B, with each node augmented to store the maximum value beneath it in the tree. To further reduce the space usage for short streams—and to avoid the initial $\Theta(|\Sigma|)$ time and space required for initialization of the array B and the tree T—we use a red/black tree in place of the

LIS-small-alphabet($\mathcal{S}$): *// compute the length of the LIS of the stream $\mathcal{S} = \langle x_1, \ldots, x_n \rangle$, with $x_i \in \Sigma$.*

// $B[j]$ stores the length of the longest increasing subsequence ending with element $j \in \Sigma$ in the stream so far.
// Treat an uninitialized $B[j]$ as $B[j] = 0$.

1. While there are elements left in the stream $\mathcal{S}$:
   (a) read the next element x_i from $\mathcal{S}$.
   (b) set $B[x_i] := 1 + \max\{B[1], \ldots, B[x_i - 1], B[x_i]\}$.
2. Return $\max\{B[1], \ldots, B[|\Sigma|]\}$.

Figure 2: An algorithm to compute the length of the LIS in stream $\mathcal{S}$ when the alphabet Σ has small size. To implement Steps 1b and 2 efficiently, we use a balanced binary tree T to store the initialized values of $B[j]$. Every internal node u of T is augmented to store the maximum leaf-node value in the subtree rooted at u. This augmentation allows the computation of any "prefix maximum"—that is, $\max\{B[1], \ldots, B[j]\}$ for any index j—in $O(\log |\Sigma|)$ time, and the augmented data can be updated in $O(\log |\Sigma|)$ time when $B[x_i]$ is initialized (i.e., inserted into T) or increased (i.e., deleted and reinserted into T).

complete binary tree, again augmenting every node to store the maximum value beneath it in the tree. When the element $a \in \Sigma$ first appears in the stream, we insert the key/value pair $(a, B[a])$ into the tree; when $B[x_i]$ is updated we delete and reinsert the x_i entry from the tree. The maximum value beneath each node in the tree can be maintained through node deletion and insertion in a red/black tree in $O(\log |T|)$ time. (This data structure is a simplified version of an order-statistic tree [10].) See Figure 2 for the pseudocode.

Theorem 3.3. *We can decide whether the LIS of a stream of integers drawn from alphabet Σ has length at least some given number k, or compute the length k of the LIS of the given stream, with a one-pass streaming algorithm that uses $O(|\Sigma| \log k)$ space and has update time $O(\log |\Sigma|)$.*

Proof. Correctness of the while loop in LIS-small-alphabet($\cdot$) follows immediately by induction, and the space requirement and update time follow immediately from the balance of the tree T and the above discussion of the data augmentation. □

When $|\Sigma|$ is constant, this algorithm requires only $O(\log k)$ space and $O(1)$ update time. Note also that if the alphabet size $|\Sigma|$ is unknown, we achieve the same update time and space usage as described above; this algorithm does not require knowledge of $|\Sigma|$ to compute the LIS.

3.2 Finding an LIS

The algorithms described in the previous section only compute the length of the LIS, but do not explicitly find such a sequence. We can straightforwardly modify Fredman's algorithm to return an LIS itself: as the stream elements are read, we maintain, for each ℓ, a length-ℓ increasing sequence σ^ℓ whose last element is $A[\ell]$:

- initialize $\sigma^0 := \langle\rangle$ to be the empty sequence in Step 1.

- when $A[\ell+1]$ is set to x_i in Step 2c because $A[\ell] \le x_i < A[\ell+1]$, also update the sequence $\sigma^{\ell+1} := \mathsf{concat}(\sigma^\ell, \langle x_i \rangle)$.
- return $\sigma^{k'}$ instead of k' in Step 3.

Updating the sequence $\sigma^{\ell+1}$ can be done in $O(1)$ time using a linked node structure. Specifically, maintain an array of pointers $P[1 \ldots k']$. For each new data item x_i, create a node with key x_i. Call the new node η. To update $\sigma^{\ell+1}$ to be $\mathsf{concat}(\sigma^\ell, \langle x_i \rangle)$, simply set $P[\ell+1]$ to point to η (the node for x_i), and set the pointer for node η to point to the same node as $P[\ell]$.

Updating in this way may use too much space, because we never delete any nodes. To avoid wasting this space, we use reference counting, a standard garbage-collection technique. Augment each node to include a counter of the number of pointers that point to it. Also maintain a list of nodes to be deleted. Then, when we insert the node η, we increment the counter for the node to which η points. We decrement the counter for the node to which $P[\ell+1]$ previously pointed. If the counter for that node is now zero, we add it to the list of nodes to be deleted. In any case, if the list of nodes to be deleted is nonempty, we delete one node from the list. (Of course, whenever we delete a node, we decrement the counter of the node to which it pointed. If that counter is now zero, we add that node to the list of nodes to be deleted.)

This modification adds only a constant amount of extra running time per update, so the update time per element remains $O(\log k)$ or $O(\log\log|\Sigma|)$, and the space requirement is $O(k^2 \log|\Sigma|)$.

In the remainder of this section, we present a multipass streaming algorithm based upon the ideas of Fredman's algorithm that outputs an LIS of length k using $O(k^{1+\varepsilon}\log|\Sigma|)$ space in $\lceil \log(1+1/\varepsilon) \rceil$ passes over the data, for any $\varepsilon > 0$. (We first describe a two-pass streaming algorithm that requires less space than $\mathsf{Fredman}(\cdot)$, and we will subsequently generalize this algorithm to a p-pass algorithm for a general p.) The space used by this algorithm is nearly optimal, because merely storing a length-k sequence in general requires $\Omega(k \log|\Sigma|)$ space when $|\Sigma| = \Omega(n)$ and when n is sufficiently larger than k.

A two-pass algorithm. Note that Fredman's algorithm maintains k sequences $\sigma^1, \ldots, \sigma^k$, taking a total of $O(k^2 \log|\Sigma|)$ space. The key modification for our two-pass algorithm is the following: during the first pass over the data, the algorithm only remembers part of each σ^ℓ; specifically, we store every qth element of σ^ℓ—for a value of q to be computed below—plus the last element of σ^ℓ. That is, for each length $\ell \in \{1, \ldots, k\}$, we maintain the sequence

$$\widetilde{\sigma}^\ell = \langle \sigma^\ell_1, \sigma^\ell_{q+1}, \sigma^\ell_{2q+1}, \ldots, \sigma^\ell_{\lfloor \frac{\ell-2}{q} \rfloor q+1}, \sigma^\ell_\ell \rangle,$$

where σ^ℓ is a length-ℓ increasing sequence ending with $A[\ell]$, as in the one-pass version of the algorithm. Storing the last element of $\widetilde{\sigma}^\ell$ will give us all of the information that we need to update the stored sequences. Recall that $\mathsf{all\text{-}but\text{-}last}(\sigma)$ denotes the sequence σ with its last element removed, and that $\sigma^0 := \langle\rangle$ is initialized to the empty sequence. The update rule during the first pass of the algorithm is then the following:

$$\text{when } A[\ell] \le x_i < A[\ell+1]\text{:} \qquad \widetilde{\sigma}^{\ell+1} := \begin{cases} \mathsf{concat}(\widetilde{\sigma}^\ell, \langle x_i \rangle) & \text{if } \ell \equiv 1 \pmod{q} \\ \mathsf{concat}(\mathsf{all\text{-}but\text{-}last}(\widetilde{\sigma}^\ell), \langle x_i \rangle) & \text{otherwise.} \end{cases}$$

That is, the sequence $\sigma^{\ell+1}$ is just σ^ℓ with the next stream element put either after the last element of σ^ℓ (if $\ell+1$ is a qth index) or in place of the last element of σ^ℓ (if not).

After the first pass is complete, we discard the subsequences $\widetilde{\sigma}^1, \ldots, \widetilde{\sigma}^{k-1}$. Thus the only information we retain is the subsequence

$$\widetilde{\sigma}^k = \langle \sigma^k_1, \sigma^k_{q+1}, \sigma^k_{2q+1}, \ldots, \sigma^k_{\lfloor \frac{k-2}{q} \rfloor q+1}, \sigma^k_k \rangle$$

where σ^k is a length-k LIS of the input. For ease of notation in the following, we will write the elements of $\widetilde{\sigma}^k$ as an array z, so that $z[i] := \widetilde{\sigma}^k_i = \sigma^k_{iq+1}$ for $i \in \{1, \ldots, \lfloor (k-2)/q \rfloor\}$ and $z[\lfloor (k-2)/q \rfloor + 1] := \widetilde{\sigma}^k_{\lfloor (k-2)/q \rfloor + 1} = \sigma^k_k$.

In the second pass, we want to "fill in the blanks" of the subsequence $\widetilde{\sigma}^k$ to produce σ^k. Specifically, we want to find an increasing subsequence τ^ℓ that starts with $z[\ell]$ and ends with $z[\ell+1]$, for each index ℓ. Notice that we can do this sequentially—for one ℓ at a time—because two consecutive τ subsequences do not overlap except at the endpoints. Each desired subsequence τ^ℓ has length exactly $q+1$, except for the last subsequence (which has length at most $q+1$). The LIS of the stream is formed by simply concatenating the τ sequences, without duplicating the boundary elements $z[\ell]$ that appear in the consecutive $\tau^{\ell-1}$ and τ^ℓ sequences.

We claim that the space usage of this two-pass algorithm is better than Fredman's algorithm. As usual, let k denote the length of the stream's LIS. In the first pass, we store a sequence of ℓ/q elements of Σ for each $\ell \in \{1, \ldots, k\}$. Overall, then, the space consumption of the first pass is $\sum_{\ell=1}^{k} (\ell/q)(\log |\Sigma|) = O((\frac{k^2}{q}) \log |\Sigma|)$. In the second pass, we use $O(q^2 \log |\Sigma|)$ space to compute the length-$(q+1)$ sequences τ and $O(k \log |\Sigma|)$ space for the final LIS. Thus the total space required for the entire second pass is $O(q^2 \log |\Sigma| + k \log |\Sigma|)$. The total space requirement of the algorithm, then, is $O(\max(\frac{k^2}{q}, q^2) \cdot \log |\Sigma| + k \log |\Sigma|)$. This quantity is minimized by selecting $q := k^{2/3}$, which makes the overall space consumption $O(k^{4/3} \log |\Sigma|)$.

Generalizing to a p-pass algorithm. We can generalize this idea to a larger number of passes by computing each subsequence τ^ℓ recursively. As before, in the first pass the algorithm stores only every qth element in each σ^ℓ, and then all stored subsequences except $\widetilde{\sigma}^k$ are discarded. Then the algorithm uses $p-1$ passes to find the $\Theta(k/q)$ subsequences $\tau^1, \tau^2, \ldots, \tau^{\lfloor (k-2)/q \rfloor}$, where each subsequence τ^ℓ has length $O(q)$. The pseudocode is shown in Figure 3.

Let $S(k, p)$ denote the space required by a p-pass algorithm to find a subsequence of length k. Then we have the following recurrence: $S(k, p) = \max(O((\frac{k^2}{q}) \log |\Sigma|), S(q, p-1)) + O(k \log |\Sigma|)$. Hence, the space requirements are optimized by setting $q := k^{1-1/(2^p-1)}$, which gives $S(k, p) = O(k^{1+1/(2^p-1)} \log |\Sigma|)$.

Theorem 3.4. *Fix a parameter $\varepsilon > 0$. For a given length k, we can find a length-k increasing subsequence of a stream of integers drawn from Σ with a $\lceil \log(1 + 1/\varepsilon) \rceil$-pass streaming algorithm that uses $O(k^{1+\varepsilon} \log |\Sigma|)$ space and has update time $O(\log k)$ or $O(\log \log |\Sigma|)$.*

We can find the LIS of a stream even when its length k is not known in advance, using the same number of passes, the same update time, and space $O(\frac{1}{\varepsilon} k^{1+\varepsilon} \log |\Sigma|)$.

Proof. Given a parameter $\varepsilon > 0$, we set $p := \lceil \log(1 + 1/\varepsilon) \rceil$. Then the p-pass streaming algorithm multipassLIS$(\mathcal{S}, p, k)$ uses space $O(k^{1+\varepsilon} \log |\Sigma|)$ to compute the LIS of the given stream $\mathcal{S}$. Correctness follows just as in Fredman's algorithm, and the space bound was derived above.

When k is unknown, we can achieve the stated bounds via a slight modification to multipassLIS(). Define a recursive sequence by $q_0 := 1$ and $q_{j+1} := q_j + q_j^{1-\varepsilon}$ for all $j \geq 0$. In the first pass of the

multipassLIS$(\mathcal{S}, p, k)$ *// compute a length-k LIS of the stream $\mathcal{S} = \langle x_1, \ldots, x_n \rangle$ using p passes over the data. // Maintain every qth element of $\mathcal{S}$'s LIS; each pass successively decreases q to include more and more elements.*

1. Set $q := k^{1-1/(2^p-1)}$.
2. Return fill-in-the-blanks$(\mathcal{S}, \text{create-blanks}(\mathcal{S}, k, q), p-1, q)$.

create-blanks$(\mathcal{S}, k, q)$: *// Find every qth entry of a length-k increasing sequence in $\mathcal{S}$.*

1. Initialize $\widetilde{\sigma}^0 := \langle\rangle$. *// Treat uninitialized $\widetilde{\sigma}_j^\ell$ as infinite.*
2. While there are still elements left in $\mathcal{S}$:
   (a) read the next element x_i from $\mathcal{S}$.
   (b) find the ℓ such that $\widetilde{\sigma}_\ell^\ell \le x_i < \widetilde{\sigma}_{\ell+1}^{\ell+1}$ and then set

$$\widetilde{\sigma}^{\ell+1} := \begin{cases} \text{concat}(\widetilde{\sigma}^\ell, \langle x_i \rangle) & \text{if } \ell \equiv 1 \pmod{q} \\ \text{concat}(\text{all-but-last}(\widetilde{\sigma}^\ell), \langle x_i \rangle) & \text{otherwise.} \end{cases}$$

3. Return $\widetilde{\sigma}^k$.

fill-in-the-blanks$(\mathcal{S}, z, p, k)$: *// Use p passes to fill in a length-k increasing sequence between each z_j, z_{j+1}.*

1. If $k = 1$ or $p = 0$, then return z.
2. Initialize $Z' := \langle\rangle$ and $q := k^{1-1/(2^p-1)}$.
3. Repeatedly read the next element from $\mathcal{S}$ until it equals z_1.
4. For $j = 1$ to $|z| - 1$:
   (a) initialize $\widetilde{\sigma}^1 := \langle z_j \rangle$ and reset all other $\widetilde{\sigma}^\ell$ to be uninitialized.
   (b) repeat until $\widetilde{\sigma}_k^k = z_{j+1}$ or there are no elements remaining in $\mathcal{S}$:
      i. read the next element x_i from $\mathcal{S}$.
      ii. if $x_i \ge z_j$ then find the ℓ such that $\widetilde{\sigma}_\ell^\ell \le x_i < \widetilde{\sigma}_{\ell+1}^{\ell+1}$ *// Treat uninitialized $\widetilde{\sigma}_j^\ell$ as infinite.* and then set

$$\widetilde{\sigma}^{\ell+1} := \begin{cases} \text{concat}(\widetilde{\sigma}^\ell, \langle x_i \rangle) & \text{if } \ell \equiv 1 \pmod{q} \\ \text{concat}(\text{all-but-last}(\widetilde{\sigma}^\ell), \langle x_i \rangle) & \text{otherwise.} \end{cases}$$

   (c) set $Z' := \text{concat}(Z', \text{all-but-last}(\widetilde{\sigma}^{k^*}))$, where k^* denotes the largest index such that $\widetilde{\sigma}_{k^*}^{k^*} = z_{j+1}$. (Unless there are no more elements in $\mathcal{S}$, the quantity k^* is simply $k^* = k$. If the stream has been read entirely, then k^* may be smaller.)
5. Set $Z' := \text{concat}(Z', \langle z_{|z|} \rangle)$.
6. Return fill-in-the-blanks$(\mathcal{S}, Z', p-1, q)$.

Figure 3: A p-pass streaming algorithm to find a longest increasing subsequence. This algorithm uses space $O(k^{1+1/(2^p-1)} \log |\Sigma|)$ and has update time $O(\log k)$ or $O(\log \log |\Sigma|)$.

algorithm only, for each length ℓ we maintain the sequence $\widetilde{\sigma}^\ell = \langle \sigma^\ell_{\lceil q_0 \rceil}, \sigma^\ell_{\lceil q_1 \rceil}, \sigma^\ell_{\lceil q_2 \rceil}, \ldots, \sigma^\ell_{\lceil q_{t_\ell} \rceil}, \sigma^\ell_\ell \rangle$ where t_ℓ is the largest index such that $\lceil q_{t_\ell} \rceil < \ell$. The update rule then changes to the following:

$$\widetilde{\sigma}^{\ell+1} := \begin{cases} \mathsf{concat}(\widetilde{\sigma}^\ell, \langle x_i \rangle) & \text{if } \ell = \lceil q_j \rceil \text{ for some } j \\ \mathsf{concat}(\mathsf{all\text{-}but\text{-}last}(\widetilde{\sigma}^\ell), \langle x_i \rangle) & \text{otherwise.} \end{cases}$$

After the first pass is complete, we again discard all stored sequences except the last. That is, we retain only the sequence

$$\widetilde{\sigma}^k = \langle \sigma^k_{\lceil q_0 \rceil}, \sigma^k_{\lceil q_1 \rceil}, \sigma^k_{\lceil q_2 \rceil}, \ldots, \sigma^k_{\lceil q_t \rceil}, \sigma^k_k \rangle$$

where t is the largest index such that $\lceil q_t \rceil < k$. By the definition of the q_j's, we see that $\lceil q_{j+1} \rceil - \lceil q_j \rceil \leq q_{j+1} - q_j + 1 \leq q_j^{1-\varepsilon} + 1 \leq k^{1-\varepsilon}$, for all $j + 1 \leq t$. So the number of elements from the LIS that we are missing, between any $\sigma^k_{\lceil q_j \rceil}$ and $\sigma^k_{\lceil q_{j+1} \rceil}$, is at most $k^{1-\varepsilon}$. But in multipassLIS() when k is known in advance the size of the gap between elements that are retained after the first pass is also $k^{1-\varepsilon}$. Thus the size of the gaps in the retained sequence in the modified algorithm is no larger than the size of the gaps in the standard algorithm, and we can resume the standard algorithm for the second pass and all passes that follow, with a slight modification to fill-in-the-blanks() to handle the fact that the gaps are now of varying width. Correctness follows analogously to the standard algorithm.

Thus the space usage and update time are identical to multipassLIS() for all passes after the first. For the first pass of the algorithm, the update time is identical to the standard algorithm. However, the space used is $O(kt \log |\Sigma|)$. We now bound t. First, notice that the q_j's form an increasing sequence, and hence the definition gives us that

$$q_{j+j'} \geq q_j + j' q_j^{1-\varepsilon} \qquad \text{for any } j, j' \geq 0. \tag{1}$$

We now claim the following:

$$\text{for all integers } r \geq 1\text{: if } q_\ell < 2^r, \text{ then } \ell \leq \frac{2^{r\varepsilon}}{\varepsilon \ln 2} + r \tag{2}$$

We prove (2) by induction on r. The base case $r = 1$ is immediate. For the inductive case, assume $q_\ell < 2^r$, and let ℓ' be the index such that $q_{\ell'} < 2^{r-1} \leq q_{\ell'+1}$. By the inductive hypothesis, then, we have that $\ell' \leq \frac{2^{(r-1)\varepsilon}}{\varepsilon \ln 2} + r - 1$. Hence we have

$$\begin{array}{rcll} 2^r & > & q_\ell & \text{by assumption} \\ & \geq & q_{\ell'+1} + (\ell - \ell' - 1) q_{\ell'+1}^{1-\varepsilon} & \text{by (1)} \\ & \geq & 2^{r-1} + (\ell - \ell' - 1) \left(2^{r-1}\right)^{1-\varepsilon} & \text{by the fact that } q_{\ell'+1} \geq 2^{r-1}\text{, by definition of } \ell'. \end{array}$$

Solving for ℓ, we have

$$\begin{array}{rcll} \ell & \leq & 2^{(r-1)\varepsilon} + 1 + \ell' & \\ & \leq & 2^{(r-1)\varepsilon} + 1 + \frac{2^{(r-1)\varepsilon}}{\varepsilon \ln 2} + r - 1 & \text{by the inductive hypothesis} \\ & \leq & \frac{2^{(r-1)\varepsilon}}{\varepsilon \ln 2} (1 + \varepsilon \ln 2) + r & \text{by algebraic manipulation.} \end{array}$$

Because $1 + x \leq e^x$ for all $x \geq 0$, we have that $1 + \varepsilon \ln 2 \leq e^{\varepsilon \ln 2} = 2^\varepsilon$. Thus we see that $\ell \leq \frac{2^{r\varepsilon}}{\varepsilon \ln 2} + r$, and claim (2) follows. Setting $r = \lceil \lg k \rceil$ in (2), we see that $t = O(\frac{1}{\varepsilon} k^\varepsilon)$. Thus the total space used in the first pass of the algorithm is $O(\frac{1}{\varepsilon} k^{1+\varepsilon} \log |\Sigma|)$, as desired. □

Theorem 3.4 is our main result on computing actual longest increasing subsequences, but before we turn to lower bounds on LIS in Section 4, we note that we can again more efficiently find an LIS itself when the alphabet Σ is small. In particular, when $|\Sigma| = O(1)$, we can actually find an LIS using just $O(\log k)$ space and $O(1)$ update time, the same requirements that we had for merely determining its length.

Theorem 3.5. *We can find an LIS of a stream of integers drawn from alphabet Σ with a one-pass streaming algorithm that uses $O(|\Sigma|^2 \log k)$ space and has update time $O(\log |\Sigma|)$.*

Proof. First, observe that rather than maintaining σ^ℓ for each length ℓ as in Fredman's algorithm, we need only maintain $\sigma^{B[j]}$ for each $j \in \Sigma$. (Recall that $B[j]$ denotes the length of the longest increasing subsequence that ends with the element $j \in \Sigma$ in the stream so far. See Figure 2.) Furthermore, we only need $O(|\Sigma| \log k)$ space to store each sequence σ^ℓ, because it suffices to keep track of the indices i for which $\sigma_i^\ell \neq \sigma_{i+1}^\ell$, and there are at most $|\Sigma|$ such indices in an increasing sequence. Hence, we can find the LIS using space $O(|\Sigma|^2 \log k)$ and update time $O(\log |\Sigma|)$. □

4 Lower Bounds for LIS

We now turn our attention to establishing lower bounds on the space requirements for streaming algorithms that solve the LIS problem. In this section, we prove that $\Omega(k)$ bits of storage are required to decide if the LIS of a stream of n elements has length at least k, for any $n = \Omega(k^2)$. Our lower bounds, like most lower bounds on space usage in the data-streaming model, are derived from the well-observed connection between space consumption for streaming algorithms and the communication complexity of a related two-party problem. Specifically, our proof is based on reducing the *set-disjointness problem* to LIS in the data-streaming model:

Definition 4.1 (Set Disjointness). *Party A holds an n-bit string s_A, and Party B holds another n-bit string s_B. The pair $\langle s_A, s_B \rangle$ forms a 'yes' instance for the* Set-Disjointness *problem if and only if the ith bit of both s_A and s_B is 1, for some index i.*

We say that s_A and s_B *intersect* in a 'yes' instance and are *disjoint* in a 'no' instance. The *communication complexity* of a protocol solving Set-Disjointness is the maximum number of bits communicated between Party A and Party B, taken over all valid inputs. Lower bounds for the set-disjointness problem have been studied extensively (e.g., [6, 26, 28]), and this problem is known to have high communication complexity. The strongest results show that even in the randomized setting, Set-Disjointness requires a large amount of communication:

Theorem 4.2 (Bar-Yossef, Jayram, Kumar, Sivakumar [6]). *Let $\delta \in (0, 1/4)$. Let $\mathcal{A}$ be a (possibly randomized) protocol for the* Set-Disjointness *problem that, for every input $\langle s_A, s_B \rangle$, is correct with probability at least $1 - \delta$. Then $\mathcal{A}$ requires at least $\frac{n}{4}(1 - 2\sqrt{\delta})$ bits of communication between Party A and Party B. The same bound holds even if $\mathcal{A}$ is only required to be correct when the vectors s_A and s_B both contain exactly $n/4$ ones.*

We now reduce Set-Disjointness to the problem of determining if an increasing subsequence of length $\sqrt{n}$ exists in an n-element stream. This reduction will show that deciding whether the LIS has length k requires $\Omega(k)$ space in the streaming model, even with randomization and some chance of error, whenever $n = \Omega(k^2)$.

Suppose we are given an instance $\langle s_A, s_B \rangle$ of Set-Disjointness, where $n := |s_A| = |s_B|$. We construct a stream $\mathsf{S\text{-}lis}(s_A, s_B)$ satisfying three relevant properties: (i) the first half of the stream will depend only on s_A; (ii) the second half of the stream will depend only on s_B; and (iii) the stream's LIS will have length at least $n+1$ if and only if $\langle s_A, s_B \rangle$ are non-disjoint.

For each index $i \in \{1, \ldots, n\}$, write $N_i := (n+1) \cdot (i-1)$ for readability. We associate with an index i the $(n+1)$-element sequence $\langle N_i + 1, \ldots, N_i + n, N_{i+1} \rangle$, divided into two parts.

- Define $\mathsf{A\text{-}part}(i) := \langle N_i+1, N_i+2, \ldots, N_i+i \rangle$ to be the length-i increasing sequence consisting of the first i of these integers. Define $\mathsf{B\text{-}part}(i) := \langle N_i + i + 1, N_i + i + 2, \ldots, N_{i+1} \rangle$ to be the length-$(n-i+1)$ increasing sequence consisting of the remaining $(n-i+1)$ of these integers.
- Define $\mathsf{S\text{-}lis}_A(s_A)$ to be the sequence consisting of the concatenation of the sequences $\mathsf{A\text{-}part}(i)$ for every $i \in \{i : s_A(i) = 1\}$, listed in decreasing order of the index i. Similarly, let $\mathsf{S\text{-}lis}_B(s_B)$ be the sequence consisting of the sequences $\mathsf{B\text{-}part}(i)$ for every $i \in \{i : s_B(i) = 1\}$, also listed in decreasing order of the index i.
- Define the stream $\mathsf{S\text{-}lis}(s_A, s_B) := \mathsf{concat}(\mathsf{S\text{-}lis}_A(s_A), \mathsf{S\text{-}lis}_B(s_B))$.

The first two requirements above are clearly satisfied: the sequence $\mathsf{S\text{-}lis}_A(s_A)$ depends only on the vector s_A, and similarly the sequence $\mathsf{S\text{-}lis}_B(s_B)$ depends only on the vector s_B. We claim that the third property is satisfied as well:

Lemma 4.3. *If the vectors s_A and s_B intersect, then $|\mathsf{LIS}(\mathsf{S\text{-}lis}(s_A, s_B))| \geq n+1$. If the vectors s_A and s_B do not intersect, then $|\mathsf{LIS}(\mathsf{S\text{-}lis}(s_A, s_B))| \leq n$.*

Proof. We start with the obvious direction. Suppose that s_A and s_B intersect—i.e., suppose that $s_A(i) = s_B(i) = 1$ for some particular i. Then observe that $\mathsf{S\text{-}lis}(s_A, s_B)$ contains the length-$(n+1)$ increasing subsequence $\mathsf{concat}(\mathsf{A\text{-}part}(i), \mathsf{B\text{-}part}(i)) = \langle N_i + 1, N_i + 2, \ldots, N_i + n, N_{i+1} \rangle$.

For the converse direction, we prove the contrapositive. Suppose that s_A and s_B do not intersect. Observe that for any two indices i and j such that $i < j$ we have the following two facts: (1) $\mathsf{A\text{-}part}(i)$ follows $\mathsf{A\text{-}part}(j)$ in $\mathsf{S\text{-}lis}_A(s_A)$, and (2) the integers in $\mathsf{A\text{-}part}(i)$ are all smaller than those in $\mathsf{A\text{-}part}(j)$. Thus any increasing subsequence that is wholly within $\mathsf{S\text{-}lis}_A(s_A)$ can contain integers from $\mathsf{A\text{-}part}(i)$ for only a single index i. Likewise, any increasing subsequence that is wholly within $\mathsf{S\text{-}lis}_B(s_B)$ can contain integers from $\mathsf{B\text{-}part}(j)$ for only a single index j. Thus the only potential length-$(n+1)$ increasing subsequences must be subsequences of $\mathsf{concat}(\mathsf{A\text{-}part}(i), \mathsf{B\text{-}part}(j))$ for some indices i and j so that $s_A(i) = s_B(j) = 1$. (By assumption, then, we must have $i \neq j$.) Furthermore, unless $i < j$, all the integers in $\mathsf{A\text{-}part}(i)$ are larger than the integers in $\mathsf{B\text{-}part}(j)$. Thus the LIS of $\mathsf{S\text{-}lis}(s_A, s_B)$ has length at most $|\mathsf{A\text{-}part}(i)| + |\mathsf{B\text{-}part}(j)| = i + n - j + 1 \leq n$. □

We now improve the construction so that the resulting stream $\mathsf{S\text{-}lis}(s_A, s_B)$ is a *permutation*, i.e., it contains each element of $\{0, 1, \ldots, |\Sigma| - 1\}$ exactly once. (Previously our alphabet has been $\Sigma = \{1, \ldots, |\Sigma|\}$; here we use a zero-indexed alphabet because our construction is an extension of the above $\mathsf{S\text{-}lis}(s_A, s_B)$, and adding a zero element to the alphabet allows us to give a much simpler description of the extension.) We will show that a suitable $|\Sigma| = \Theta(n^2)$ suffices.

We modify $\mathsf{S\text{-}lis}_A(s_A)$ and $\mathsf{S\text{-}lis}_B(s_B)$ as follows: we include the integers from $\mathsf{A\text{-}part}(i)$ and $\mathsf{B\text{-}part}(i)$ even when $s_A(i) = 0$ or $s_B(i) = 0$, but we include them in such a way that only two of these elements can be part of a longest increasing subsequence.

- Let $U_A := \{x \mid \exists i : s_A(i) = 0, x \in \mathsf{A\text{-}part}(i)\}$ and let $U_B := \{x \mid \exists i : s_B(i) = 0, x \in \mathsf{B\text{-}part}(i)\}$ denote the sets of "unused" numbers in $\mathsf{S\text{-}lis}_A(s_A)$ and $\mathsf{S\text{-}lis}_B(s_B)$, respectively—that is, those integers in $\{1, \ldots, N_{n+1}\}$ that do not appear in the stream $\mathsf{S\text{-}lis}(s_A, s_B)$.

- Define $\mathsf{pad\text{-}A}(s_A)$ to be the sequence consisting of integers in U_A listed in decreasing order, followed by the integer 0. Define $\mathsf{pad\text{-}B}(s_B)$ to be the sequence consisting first of the lone integer $N_{n+1} + 1 = (n+1) \cdot n + 1$, followed by the integers in U_B listed in decreasing order.

- Now define $\mathsf{S\text{-}lis}^\pi_A(S_A) := \mathsf{concat}(\mathsf{pad\text{-}A}(s_A), \mathsf{S\text{-}lis}_A(s_A))$. (We write '$\pi$' in the superscript to denote a permutation.) Similarly, define $\mathsf{S\text{-}lis}^\pi_B(S_B) := \mathsf{concat}(\mathsf{S\text{-}lis}_B(s_B), \mathsf{pad\text{-}B}(s_B))$.

- Finally, define $\mathsf{S\text{-}lis}^\pi(s_A, s_B) := \mathsf{concat}(\mathsf{S\text{-}lis}^\pi_A(s_A), \mathsf{S\text{-}lis}^\pi_B(s_B))$. This stream consists of the "missing" elements of s_A in decreasing order, followed by 0, then followed by the "present" elements; then the "present" elements of s_B, followed by $(n+1) \cdot n + 1$, followed by the "missing" elements of s_B in decreasing order.

It is straightforward to verify that $\mathsf{S\text{-}lis}^\pi(s_A, s_B)$ is a permutation of the set $\{0, \ldots, (n+1) \cdot n + 1\}$. Furthermore, it is easy to see that the additions of $\mathsf{pad\text{-}A}(s_A)$ and $\mathsf{pad\text{-}B}(s_B)$ each increase the LIS by exactly one:

Lemma 4.4. *If the vectors s_A and s_B intersect, then $|\mathsf{LIS}(\mathsf{S\text{-}lis}^\pi(s_A, s_B))| \geq n+3$. If the vectors s_A and s_B do not intersect, then $|\mathsf{LIS}(\mathsf{S\text{-}lis}^\pi(s_A, s_B))| \leq n + 2$.*

Proof. The stream $\mathsf{S\text{-}lis}^\pi(s_A, s_B)$ consists of three segments: (i) a prefix ending with the element 0 that is a decreasing sequence, (ii) the stream $\mathsf{S\text{-}lis}(s_A, s_B)$, and (iii) a suffix starting with the element $(n+1) \cdot n + 1$ that is again a decreasing sequence. Thus any increasing subsequence of $\mathsf{S\text{-}lis}^\pi(s_A, s_B)$ can contain at most one element from the prefix segment and at most one element from the suffix. Thus the following sequence must be a longest increasing subsequence of $\mathsf{S\text{-}lis}^\pi(s_A, s_B)$: first the integer 0, then an LIS of $\mathsf{S\text{-}lis}(s_A, s_B)$, and finally the integer $(n+1) \cdot n + 1$. By Lemma 4.3, then, the length of the LIS of $\mathsf{S\text{-}lis}^\pi(s_A, s_B)$ is $n + 3$ if and only if the vectors s_A and s_B intersect. □

With this construction in hand, we are now ready to prove our main lower-bound result for computing longest increasing subsequences in the data-streaming model:

Theorem 4.5. *Fix any length n and any length k such that $n \geq (k-2)(k-3)+1$. Any randomized streaming algorithm $\mathcal{A}$ that decides whether $\mathsf{LIS}(\mathcal{S}) \geq k$ for any stream $\mathcal{S}$ that is a permutation of $\{0, \ldots, n\}$ with probability greater than $3/4$ requires $\Omega(k)$ space.*

Proof. Suppose that the randomized streaming algorithm $\mathcal{A}$ can decide with probability greater than $3/4$ whether any n-element permutation contains an increasing subsequence of length k. We show how to solve an instance $\langle s_A, s_B \rangle$ of the Set-Disjointness problem with $|s_A| = k - 3 = |s_B|$ with probability greater than $3/4$ by calling $\mathcal{A}$.

Specifically, the stream that we consider is $\mathcal{S} := \mathsf{concat}(\mathsf{ExtraNumbers}, \mathsf{S\text{-}lis}^\pi(s_A, s_B))$, where $\mathsf{ExtraNumbers} := \langle n-1, n-2, \ldots, (k-2)(k-3)+2 \rangle$. Observe that the LIS of $\mathcal{S}$ has exactly the same length as the LIS of $\mathsf{S\text{-}lis}^\pi(s_A, s_B)$, because the elements of $\mathsf{ExtraNumbers}$, the prepended part of $\mathcal{S}$, are all larger than those in $\mathsf{S\text{-}lis}^\pi(s_A, s_B)$, and the numbers in $\mathsf{ExtraNumbers}$ are presented in descending order. Thus, by Lemma 4.4, the LIS of $\mathcal{S}$ has length k—and $\mathcal{A}(\mathcal{S})$ returns true with probability greater than $3/4$—if and only if s_A and s_B do not intersect.

A lower bound on the space required by $\mathcal{A}$ then follows via the high communication complexity of Set-Disjointness: to solve the given instance $\langle s_A, s_B \rangle$ of the Set-Disjointness problem, Party A simulates the algorithm $\mathcal{A}$ on the stream $\mathsf{concat}(\mathsf{ExtraNumbers}, \mathsf{S\text{-}lis}^{\pi}_{A}(s_A))$ and then sends all stored information to Party B, who continues to simulate $\mathcal{A}$ on $\mathsf{S\text{-}lis}^{\pi}_{B}(s_B)$—i.e., the remainder of the stream $\mathcal{S}$. This protocol allows Party A and Party B to decide whether the LIS of $\mathcal{S}$ is at least k, and therefore whether s_A and s_B intersect. By Theorem 4.2, then, Party A must transmit at least $\Omega(k)$ bits in this protocol, and thus $\mathcal{A}$ must use $\Omega(k)$ space. □

5 Longest Common Subsequence

We now turn to the problem of finding the longest common subsequence of two streams $\mathcal{S}_1$ and $\mathcal{S}_2$ of integers. Throughout this section, we consider the *adversarial* streaming model, where elements from the two streams can be presented in any order of interleaving. In the lower bounds that we derive in this paper, the streaming algorithm will be presented with all n_1 of the elements of the stream $\mathcal{S}_1$ before it receives any of the n_2 elements of the second stream $\mathcal{S}_2$.

Let $n := \max\{n_1, n_2\}$ denote the size of the larger stream. As with all streaming problems, there is a trivial streaming algorithm that uses $\Theta(n \log |\Sigma|)$ space: simply store both n-element streams in their entirety and then run a standard (non-streaming) LCS algorithm.

There is also a trivial $|\Sigma|$-approximation working in $O(|\Sigma| \log n)$ space and $O(1)$ update time. For each element in the alphabet Σ, simply calculate the length of the LCS of the two streams using only the given element. That is, for each element $a \in \Sigma$, let $count_a$ denote the number of times that a appears in the stream in which it is less prevalent—i.e., the quantity $count_a$ denotes the minimum over $i \in \{1, 2\}$ of the number of times that the element a appears in $\mathcal{S}_i$. (The $count_a$ values can be maintained using $\Theta(|\Sigma|)$ counters, each using $O(\log n)$ space, that are incremented as elements from the stream are read.) The maximum of the values $count_a$ is clearly a lower bound on the length of the true LCS of the two streams using all elements. Furthermore, it is also within a $|\Sigma|$-factor of the optimal LCS, because any length-k sequence of elements drawn from Σ must contain at least $k/|\Sigma|$ copies of at least one of the alphabet symbols, by the pigeonhole principle. When $|\Sigma| = O(1)$, we then have a streaming algorithm that is a constant-factor approximation for LCS and uses only logarithmic space.

We can give another algorithmic upper bound for a version of LCS, based upon a simple connection with LIS. Suppose that we are first given one *reference permutation* $\mathcal{R}$, and then we are subsequently given a large number of *test permutations* $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_q$; we want to compute the LCS of $\mathcal{R}$ and $\mathcal{S}_i$ for every $1 \leq i \leq q$. Our streaming algorithm stores the permutation $\mathcal{R}$ as a lookup table and then, for each $\mathcal{S}_i$, runs the LIS algorithm from Section 3, where we treat x as less than y if and only if x appears before y in $\mathcal{R}$. This algorithm requires $O(n \log |\Sigma|)$ total space—$O(n \log |\Sigma|)$ to store $\mathcal{R}$ and $O(k \log |\Sigma|) = O(n \log |\Sigma|)$ for the LIS computation. Note that this bound is independent of q.

In the remainder of this section, we present several lower bounds for LCS, again using a series of reductions from the Set-Disjointness problem.

5.1 Lower Bound on Exact and Approximate LCS

To begin, we do not impose the restriction that the streams $\mathcal{S}_1$ and $\mathcal{S}_2$ are permutations of each other. In this setting, it is straightforward to show that $\Omega(n)$ space is required, even if we are

willing to settle for an approximation to the LCS:

Theorem 5.1. *Fix any desired approximation ratio $\rho \geq 1$. Let $\mathcal{A}$ be any randomized streaming algorithm that computes a ρ-approximation of the LCS of any two streams $\mathcal{S}_1$ and $\mathcal{S}_2$ that have length at least n, presented in adversarial order, with probability greater than $3/4$. Then $\mathcal{A}$ requires $\Omega(n)$ space, even if the algorithm is presented with all of $\mathcal{S}_1$ followed by all of $\mathcal{S}_2$.*

Proof. Suppose that the algorithm $\mathcal{A}$ can always distinguish the following two cases with probability greater than $3/4$: either streams $\mathcal{S}_1$ and $\mathcal{S}_2$ contain an LCS of length at least one, or their LCS has length zero. Thus, if $\mathcal{A}(\mathcal{S}_1, \mathcal{S}_2)$ outputs the correct answer within any approximation ratio $\rho \geq 1$, then the algorithm must be able to distinguish between the length-0 and length-1 cases, independent of the ratio ρ.

We show how to solve any instance $\langle s_A, s_B \rangle$ of Set-Disjointness with probability greater than $3/4$ with $|s_A| = 4n = |s_B|$, where s_A and s_B both contain exactly n ones, by using $\mathcal{A}$. Let the n-element stream $\mathcal{S}_1$ consist of all indices i such that $s_A(i) = 1$, listed in arbitrary order. Similarly, let $\mathcal{S}_2$ consist of all n indices i such that $s_B(i) = 1$, again listed in arbitrary order. Thus $\mathsf{LCS}(\mathcal{S}_1, \mathcal{S}_2) \geq 1$ if there is at least one index i such that both $s_A(i) = 1$ and $s_B(i) = 1$ and $\mathsf{LCS}(\mathcal{S}_1, \mathcal{S}_2) = 0$ otherwise.

This fact implies the desired lower bound, because we can solve the Set-Disjointness using $\mathcal{A}$ as in Theorem 4.5. The first party simulates $\mathcal{A}$ on the stream $\mathcal{S}_1$ and then passes its state to the second party. The second party finishes simulating $\mathcal{A}$ on the stream $\mathcal{S}_2$. By Theorem 4.2, this transmitted state must therefore use $\Omega(n)$ space. To show that we still require $\Omega(n)$ space when one or both of the streams has length strictly larger than n, we simply add arbitrary unique elements to each of the above streams. □

Although this construction establishes hardness for multiplicative approximation of LCS, a simple variant shows that any data-streaming algorithm solving LCS within an additive α requires $\Omega(n/\alpha)$ space: simply repeat each element in the streams $2\alpha + 1$ times.

5.2 Lower Bound on Exact LCS for Permutations

We now improve the construction to show a lower bound on the space required for LCS even when $\mathcal{S}_1$ and $\mathcal{S}_2$ are both permutations of the set $\{1, \ldots, n\}$.

Given an instance $\langle s_A, s_B \rangle$ of the Set-Disjointness problem where there are exactly $n/4$ ones in both s_A and s_B, we construct two streams as follows:

- Define R_A as the sequence that contains $\{i : s_A(i) = 1\}$ listed in increasing order, and define $\overline{R}_A$ as the sequence that contains $\{i : s_A(i) = 0\}$ listed in decreasing order. Similarly, define R_B to contain $\{i : s_B(i) = 1\}$ listed in increasing order, and define $\overline{R}_B$ to contain the elements of $\{i : s_B(i) = 0\}$ listed in decreasing order.
- Define $\mathsf{S\text{-}lcs}^{\pi}_A(s_A) := \mathsf{concat}(R_A, \overline{R}_A)$ and $\mathsf{S\text{-}lcs}^{\pi}_B(s_B) := \mathsf{concat}(R_B, \overline{R}_B)$.

Lemma 5.2. *If the vectors s_A and s_B intersect, then $|\mathsf{LCS}(\mathsf{S\text{-}lcs}^{\pi}_A(s_A), \mathsf{S\text{-}lcs}^{\pi}_B(s_B))| \geq n/2 + 2$. Otherwise, the length of the LCS is at most $n/2 + 1$.*

Proof. Suppose that s_A and s_B intersect. Then we claim that we can construct a common subsequence of $\mathsf{S\text{-}lcs}^{\pi}_A(s_A)$ and $\mathsf{S\text{-}lcs}^{\pi}_B(s_B)$ of length at least $n/2+2$, as follows. First choose the common

element from R_A and R_B. Because s_A and s_B intersect, the set $\{i : s_A(i) = s_B(i) = 0\}$ must contain at least $n/2+1$ elements, because there are exactly $n/4$ ones in each s_A and s_B. This fact implies the existence of a common subsequence of $\overline{R}_A$ and $\overline{R}_B$ of length at least $n/2+1$. Therefore there must be a common subsequence of $\mathsf{S\text{-}lcs}^{\pi}_A(s_A)$ and $\mathsf{S\text{-}lcs}^{\pi}_B(s_B)$ with total length $n/2+2$.

Conversely, suppose that s_A and s_B have no common element. Then none of the elements in R_A match any of the elements of R_B. Of course, some elements in R_A might be matched with elements in $\overline{R}_B$. But the elements of R_A are listed in increasing order, while the elements of $\overline{R}_B$ are listed in decreasing order. Thus, at most one element of R_A can be matched with $\overline{R}_B$. If we do take such an element as part of an LCS, then the remaining sequence must be a common subsequence of $\overline{R}_A$ and $\overline{R}_B$. But these sequences contain exactly $n/2$ common elements. So the longest sequence is length at most $n/2+1$.

On the other hand, if we do not use any common elements of R_A and $\overline{R}_B$, we can instead use common elements from R_B and $\overline{R}_A$. But in this case, we can only use one such element, because R_B is an increasing sequence, while $\overline{R}_A$ is strictly decreasing. The rest of the LCS must be a common subsequence of $\overline{R}_A$ and $\overline{R}_B$. As before, $\overline{R}_A$ and $\overline{R}_B$ have exactly $n/2$ elements in common, giving an LCS whose total length is $n/2+1$. Thus $\mathsf{LCS}(\mathsf{S\text{-}lcs}^{\pi}_A(s_A), \mathsf{S\text{-}lcs}^{\pi}_B(s_B))$ can have length at most $n/2+1$. □

Theorem 5.3. *Fix any length n and any length k such that $n \geq 2k-4$. Any randomized streaming algorithm $\mathcal{A}$ that decides whether $\mathsf{LCS}(\mathcal{S}_1, \mathcal{S}_2) \geq k$ for any streams $\mathcal{S}_1, \mathcal{S}_2$ that are permutations of $\{1, \ldots, n\}$ with probability greater than $3/4$ requires $\Omega(k)$ space.*

Proof. The theorem follows analogously to Theorem 5.1 when $n = 2k-4$: deciding whether $\mathsf{S\text{-}lcs}^{\pi}_A(s_A)$ and $\mathsf{S\text{-}lcs}^{\pi}_B(s_B)$ have a common subsequence of length $n/2+2 = k$ requires $\Omega(k) = \Omega(n)$ space, by Lemma 5.2, which together with Theorem 4.2 implies the stated lower bound.

To prove the same bounds when $n > 2k-4$, we pad the streams as in Theorem 4.5. Add the decreasing sequence $n, n-1, n-2, \ldots, 2k-4+1$ to the beginning of $\mathsf{S\text{-}lcs}^{\pi}_A(s_A)$, and add the increasing sequence $2k-4+1, 2k-4+2, \ldots, n$ to the end of $\mathsf{S\text{-}lcs}^{\pi}_B(s_B)$. Then any common subsequences of these extended sequences are either (1) contained entirely in the unextended portions of $\mathsf{S\text{-}lcs}^{\pi}_A(s_A)$ and $\mathsf{S\text{-}lcs}^{\pi}_B(s_B)$ or (2) have length at most one. Then, as before, the LCS has length k if and only if s_A and s_B intersect, and thus we require $\Omega(k)$ space to compute the LCS. □

5.3 Lower Bound on Approximating LCS for Permutations

In this section, we extend the lower-bound results on the space requirements for computing the LCS of two permutations to the case of approximation algorithms.

Consider an instance $\langle s_A, s_B \rangle$ of the Set-Disjointness problem, where $|s_A| = |s_B| = n$ and where there are exactly $n/4$ ones in s_A and s_B. Suppose that we are able to ρ-approximate the LCS of two permutations, for some approximation factor $\rho > 1$. We will show how to solve the Set-Disjointness instance using the LCS approximation.

For each index $i \in \{1, \ldots, n\}$, we construct two sequences $\mathsf{apx}_\rho\mathsf{A}(i, s_A)$ and $\mathsf{apx}_\rho\mathsf{B}(i, s_B)$ so that $|\mathsf{LCS}(\mathsf{apx}_\rho\mathsf{A}(i, s_A), \mathsf{apx}_\rho\mathsf{B}(i, s_B))|$ is ρ^2 if $s_A(i) = s_B(i) = 1$ and is at most ρ otherwise. We will then appropriately assemble these sequences for each index i. For any index $i \in \{1, \ldots, n\}$, the sequences $\mathsf{apx}_\rho\mathsf{A}(i, s_A)$ and $\mathsf{apx}_\rho\mathsf{B}(i, s_B)$ have length ρ^2 and consist of exactly the same set of ρ^2 integers. We define them as follows.

- For simplicity of notation, define $Z^{i,\rho} := \{(i-1)\cdot\rho^2+1, (i-1)\cdot\rho^2+2, \ldots, (i-1)\cdot\rho^2+\rho^2\}$. Both $\mathsf{apx}_\rho\mathsf{A}(i,s_A)$ and $\mathsf{apx}_\rho\mathsf{B}(i,s_B)$ will consist of exactly this set of integers.

- Let $\mathsf{apx}_\rho\mathsf{A}(i,s_A)$ consist of the elements of $Z^{i,\rho}$ listed in increasing order if $s_A(i)=1$.

  Let $\mathsf{apx}_\rho\mathsf{A}(i,s_A)$ consist of the elements of $Z^{i,\rho}$ listed in decreasing order if $s_A(i)=0$.

- For $s_B(i)=1$, define $\mathsf{apx}_\rho\mathsf{B}(i,s_B)$ be the elements of $Z^{i,\rho}$ listed in increasing order.

  For $s_B(i)=0$, we use the more complicated *median ordering* of the elements $Z^{i,\rho}$, so that the longest increasing subsequence and the longest decreasing subsequence both have length exactly ρ. For the set $\{1,\ldots,m^2\}$, the median ordering is defined as the following sequence:

  $$\langle m, m-1, \ldots, 1; 2m, 2m-1, \ldots, m+1; \ldots; m^2, m^2-1, \ldots, m(m-1)+1\rangle.$$

  When $s_B(i)=0$, we define $\mathsf{apx}_\rho\mathsf{B}(i,s_B)$ to be the median ordering of the elements of $Z^{i,\rho}$.

- Define $\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_A(s_A) := \mathsf{concat}(\mathsf{apx}_\rho\mathsf{A}(1,s_A), \mathsf{apx}_\rho\mathsf{A}(2,s_A), \ldots, \mathsf{apx}_\rho\mathsf{A}(n,s_A))$—that is, the concatenation of the $\mathsf{apx}_\rho\mathsf{A}(i,s_A)$ sequences in increasing order of the index i.

  Define $\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_B(s_B) := \mathsf{concat}(\mathsf{apx}_\rho\mathsf{B}(n,s_B), \mathsf{apx}_\rho\mathsf{B}(n-1,s_B), \ldots, \mathsf{apx}_\rho\mathsf{B}(1,s_B))$—that is, the concatenation of the $\mathsf{apx}_\rho\mathsf{B}(i,s_B)$ sequences in decreasing order of the index i.

Lemma 5.4. *If the vectors s_A and s_B intersect, then $|\mathsf{LCS}(\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_A(s_A), \mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_B(s_B))| \geq \rho^2$. If the vectors s_A and s_B do not intersect, then the length of the LCS is at most ρ.*

Proof. Suppose that the vectors s_A and s_B intersect. Let the index i be such that $s_A(i)=s_B(i)=1$. Then the sequences $\mathsf{apx}_\rho\mathsf{A}(i,s_A)$ and $\mathsf{apx}_\rho\mathsf{B}(i,s_B)$ are identical. Hence the sequence

$$\langle (i-1)\cdot\rho^2+1, (i-1)\cdot\rho^2+2, \ldots, i\cdot\rho^2\rangle$$

has length ρ^2 and is a subsequence of both $\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_A(s_A)$ and $\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_B(s_B)$.

Conversely, suppose that the vectors s_A and s_B do not intersect. Recall that $\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_A(s_A)$ lists the sequences $\mathsf{apx}_\rho\mathsf{A}(i,s_A)$ in increasing order of index i, while $\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_B(s_B)$ lists the sequences of $\mathsf{apx}_\rho\mathsf{B}(i,s_B)$ in decreasing order of index i. Thus any common subsequence of $\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_A(s_A)$ and $\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_B(s_B)$ can only contain numbers that form a common subsequence of $\mathsf{apx}_\rho\mathsf{A}(i,s_A)$ and $\mathsf{apx}_\rho\mathsf{B}(i,s_B)$ for some particular index i. But for every index i, we have that one of the following three cases holds:

1. $s_A(i)=1$ and $s_B(i)=0$, so $|\mathsf{LCS}(\mathsf{apx}_\rho\mathsf{A}(i,s_A), \mathsf{apx}_\rho\mathsf{B}(i,s_B))| = \rho$, because one is an increasing sequence while the other is a median sequence; or

2. $s_A(i)=0$ and $s_B(i)=1$, so the LCS has length 1, because one sequence is a decreasing sequence while the other is an increasing sequence; or

3. $s_A(i)=0$ and $s_B(i)=0$, so the LCS has length ρ, because one is an increasing sequence and the other is a median sequence.

(There can be no index i where $s_A(i)=s_B(i)=1$ because, by assumption, the vectors s_A and s_B do not intersect.) Therefore the LCS of $\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_A(s_A)$ and $\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_B(s_B)$ has length at most ρ.□

Theorem 5.5. *Fix any desired approximation ratio $\rho \geq 1$. Let $\mathcal{A}$ be any randomized streaming algorithm that decides with probability greater than 3/4 whether (i) $\mathsf{LCS}(\mathcal{S}_1, \mathcal{S}_2) \geq \rho^2$ or (ii) $\mathsf{LCS}(\mathcal{S}_1, \mathcal{S}_2) \leq \rho$, for any two permutations $\mathcal{S}_1$ and $\mathcal{S}_2$ of $\{1, \ldots, n\}$, presented in adversarial order. Then $\mathcal{A}$ requires $\Omega(n/\rho^2)$ space, even if the algorithm is presented with all of the elements of $\mathcal{S}_1$ followed by all of the elements of $\mathcal{S}_2$.*

Proof. As in our previous lower-bound theorems, we show how to solve an instance $\langle s_A, s_B \rangle$ of the Set-Disjointness problem with $|s_A| = n/\rho^2 = |s_B|$ using a streaming algorithm $\mathcal{A}$ that distinguishes the cases of an LCS of length at most ρ from at LCS of length at least ρ^2. By Lemma 5.4, deciding the following question corresponds exactly to deciding whether s_A and s_B intersect: do the constructed streams $\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_A(s_A)$ and $\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_B(s_B)$ have an LCS of length (i) at least ρ^2 or (ii) at most ρ? Thus a randomized data-streaming algorithm $\mathcal{A}$ that distinguishes cases (i) and (ii) with probability greater than 3/4 can be used to solve the Set-Disjointness problem with probability greater than 3/4. The first party simulates $\mathcal{A}$ on $\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_A(s_A)$ and then passes the state of the algorithm to the second party. The second party finishes the simulation of $\mathcal{A}$ on $\mathsf{S\text{-}apx}_\rho\text{-}\mathsf{lcs}^\pi_B(s_B)$. Again, by Theorem 4.2, this protocol implies that we need $\Omega(N/\rho^2)$ space for the LCS decision procedure. □

Corollary 5.6. *To ρ-approximate the LCS of n-element permutations, we need $\Omega(n/\rho^2)$ space.*

6 Conclusion and Future Work

A classic theorem of Erdős and Szekeres [14] follows from an elegant application of the pigeonhole principle: for any sequence $\mathcal{S}$ of $n+1$ numbers, there is either an increasing subsequence of $\mathcal{S}$ of length $\sqrt{n}$ or a decreasing subsequence of $\mathcal{S}$ of length $\sqrt{n}$. One of our original motivations for looking at the LIS problem was to consider the difficulty of deciding, given a stream $\mathcal{S}$, whether (1) the length of the LIS of $\mathcal{S}$ is at least $\sqrt{|S|}$, (2) the length of the longest *decreasing* sequence is at least $\sqrt{|S|}$, or (3) both. To do this, one needs an *exact* streaming algorithm for LIS; a minor modification to the median sequence in Section 5 shows that one can have an LIS of length $\sqrt{n}$ or length $\sqrt{n}-1$ with a longest decreasing subsequence of length $\sqrt{n}$ or length $\sqrt{n}+1$, respectively.

Of course, in the streaming model one is usually interested in *approximate* algorithms using, say, polylogarithmic space. Our lower bounds for LCS show that one needs a large amount of space for any reasonable approximation. However, our lower bounds for the LIS problem say that a streaming algorithm that distinguishes between an LIS of length k and one of length $k+1$ requires $\Omega(k)$ space. It is an interesting open question whether one can use a small amount of space to approximate LIS in the streaming model.

7 Acknowledgements

We would like to thank D. Sivakumar for suggesting the problem to us, and for fruitful discussions. Thanks also to Graham Cormode, Erik Demaine, Matt Lepinski, and Abhi Shelat for helpful discussions and comments.

Part of this work was done while the authors were visiting IBM Almaden, and part of the work of the first author was done at MIT. The work of the second author was supported in part by NSF grant CCR-0098066.

References

[1] Miklós Ajtai, T. S. Jayram, Ravi Kumar, and D. Sivakumar. Approximate counting of inversions in a data stream. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 370–379, 2002.

[2] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[4] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.

[5] Arindam Banerjee and Joydeep Ghosh. Clickstream clustering using weighted longest common subsequence. In *SIAM International Conference on Data Mining Workshop on Web Mining*, 2001.

[6] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Sciences*, 68(4):702–732, 2004.

[7] Michael A. Bender, Richard Cole, Erik D. Demaine, and Martin Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 139–151, 2002.

[8] Sergei Bespamyatnikh and Michael Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76(1-2):7–11, 2000.

[9] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[10] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Cliff Stein. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2002.

[11] Graham Cormode and S. Muthukrishnan. What's new: Finding significant differences in network data streams. *Transactions on Networking*, February 2006.

[12] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[13] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Frequency estimation of Internet packet streams with limited space. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 348–360, 2002.

[14] Paul Erdős and George Szekeres. A combinatorial problem in geometry. *Compositio Mathematica*, pages 463–470, 1935.

[15] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 174–185, 1998.

[16] Joan Feigenbaum, Sampath Kannan, Martin Strauss, and Mahesh Viswanathan. An approximate L_1-difference algorithm for massive data streams. *SIAM Journal on Computing*, 32(1):131–151, 2002.

[17] Jessica H. Fong and Martin Strauss. An approximate L_p-difference algorithm for massive data streams. *Discrete Mathematics & Theoretical Computer Science*, 4(2):301–322, 2001.

[18] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11:29–35, 1975.

[19] Anna Gilbert, Sudipto Guha, Piotr Indyk, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 389–398, 2002.

[20] Sudipto Guha, Nick Koudas, and Kyuseok Shim. Data-streams and histograms. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 471–475, 2001.

[21] Sudipto Guha, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 359–366, 2000.

[22] Monika Rauch Henzinger, Prabhakar Raghavan, and Sridar Rajagopalan. Computing on data streams. Technical Report 1998-011, Digital Equipment Corporation, Systems Research Center, May 1998.

[23] Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24:644–675, 1977.

[24] J. Hunt and T. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20:350–353, 1977.

[25] Piotr Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 189–197, 2000.

[26] B. Kalyanasundaram and G. Schnitger. The probabilistic communication complexity of set intersection. *SIAM Journal on Discrete Mathematics*, 5(5):545–557, 1992.

[27] Gurmeet Manku, Sridhar Rajagopalan, and Bruce Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 426–435, 1998.

[28] A. Razborov. On the distributional complexity of disjointness. *Journal of Computer and System Sciences*, 28(2), 1984.

[29] Michael E. Saks and Xiaodong Sun. Space lower bounds for distance approximation in the data stream model. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 360–369, 2002.

[30] David Sankoff and Joseph Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.

[31] C. Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 13:179–191, 1961.

[32] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[33] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, August 1983.

[34] Hongyu Zhang. Alignment of BLAST high-scoring segment pairs based on the longest increasing subsequence algorithm. *Bioinformatics*, 19(11):1391–1396, 2003.