Finding Longest Increasing and Common Subsequences in Streaming Data

David Liben-Nowell* †
dln@theory.lcs.mit.edu

Erik Vee* ‡
env@cs.washington.edu

An Zhu* §
anzhu@cs.stanford.edu

November 26, 2003

Abstract

In this paper, we present algorithms and lower bounds for the Longest Increasing Subsequence (LIS) and Longest Common Subsequence (LCS) problems in the data streaming model.

For the problem of deciding whether the LIS of a given stream of integers drawn from $\{1, \ldots, m\}$ has length at least k, we discuss a one-pass streaming algorithm using $O(k \log m)$ space, with update time either $O(\log k)$ or $O(\log \log m)$. For the problem of returning the actual longest increasing subsequence itself, we give a $\lceil \log(1 + 1/\varepsilon) \rceil$-pass streaming algorithm with update time $O(\log k)$ or $O(\log \log m)$ that uses space $O(k^{1+\varepsilon} \log m)$, for any $\varepsilon > 0$. We also prove a lower bound of $\Omega(k)$ on the space required for any streaming algorithm for LIS, even when the input stream is a permutation of $\{1, \ldots, m\}$.

We discuss a simple LIS-based algorithm for LCS, and we also give several lower bounds on this problem, of which the strongest is the following: when the elements of two n-element streams are presented in an adversarial order, we need space $\Omega(n/\rho^2)$ to approximate the length of their LCS to within a factor of ρ, even when the two streams are permutations of each other.

1 Introduction

Longest increasing and common subsequences. Let $\mathcal{S} = x_1, x_2, \ldots, x_n$ be a sequence of n integers. A *subsequence* of $\mathcal{S}$ is a sequence $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ with $i_1 < i_2 < \cdots < i_k$. Such a subsequence is said to be *increasing* if $x_{i_1} \leq x_{i_2} \leq \cdots \leq x_{i_k}$. In this paper, we consider two fundamental 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 subsequence).
- Longest Common Subsequence (LCS). Given two sequences $\mathcal{S}$ and $\mathcal{T}$, find a maximum-length sequence x which is a subsequence of both $\mathcal{S}$ and $\mathcal{T}$ (or find the length of x).

Both LIS and LCS are fundamental combinatorial questions which have been well-studied in the computer science community [4, 6, 11, 16, 17, 22, among many others].

*Part of this work was done while the authors were visiting IBM Almaden.

†Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

‡Department of Computer Science and Engineering, University of Washington, Seattle, WA 98115.

§Department of Computer Science, Stanford University, Stanford, CA 94305. Supported in part by a GRPW fellowship from Bell Labs, Lucent Technologies and NSF Grant EIA-0137761.

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) [3] database 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 [25] has proposed filtering the results of a BLAST query with an approach that uses an LIS algorithm as a black box to assemble the 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 [8], and a straightforward LCS computation gives the value of the optimal alignment of two sequences of DNA [21].

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

The theoretical computer science community 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* [15] 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 passes. 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.

Our results: LIS and LCS in the data streaming model. In this paper, we study the difficulty of finding longest increasing subsequences and longest common subsequences in the data streaming model. We are motivated in our exploration by the fact that LIS and LCS are both fundamental combinatorial questions; 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.

One notable obstacle that we face in the LIS problem is that, unlike many problems that have been previously considered in the streaming model, 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 [7, 9], clustering streaming data [14], or finding order statistics for a given stream [2, 19]—are entirely independent of the order of the elements presented in $\mathcal{S}$; permuting the order of the items in the stream does not affect the correct answers to these questions. The problem of counting *inversions* in a stream [1]—i.e., the number of pairs of indices $\langle i, j \rangle$ such that $i < j$ but $x_i > x_j$—is an inherently order-based problem, but much more local than that of LIS in the sense that an inversion is a relation between exactly two items in the stream, whereas an increasing subsequence of length ℓ is a relation among ℓ items.

In this sense, the LIS problem is more closely aligned to estimating the histogram of the stream [12, 13]. However, the solution to the LIS may be incredibly sensitive to small changes in the data. For instance, consider an LIS that consists primarily of the same repeated value. If we change the data stream so that many occurences of this value are slightly smaller, it radically changes the LIS. Similar notions apply to LCS as well. While this does not preclude efficient streaming algorithms for LIS or LCS, it does suggest some of the difficulties.

In this paper, 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. We give a one-pass streaming algorithm that uses $O(k \log m)$ space to compute the length of the longest increasing subsequence for a given input stream, where $m \geq \max x_i$ is an upper bound on the largest element in the stream, and k is the length of the LIS. (This algorithm was also discovered independently by Fredman [11] and again by Bespamyatnikh and Segal [6], though not in the context of the data streaming model.) Our algorithm maintains values $A[1 \ldots k']$, where $A[i] \in \{1, \ldots, m\}$ is the smallest possible last element of all increasing subsequences of length i in the part of the stream that has already been read, and k' is the length of the LIS for the stream so far. As we read each element, we can update the array A in time $O(\log k)$. This algorithm can also be implemented using van Emde Boas queues or y-fast trees to achieve an update time of $O(\log \log m)$ [23, 24]. For the problem of *returning* the length-k LIS of a given stream, we give a one-pass streaming algorithm that uses $O(k^2 \log m)$ space. In the context of multipass streaming algorithms, we reduce the space requirement to $O(k^{1+\varepsilon} \log m)$ by using $\lceil \log(1 + 1/\varepsilon) \rceil$ passes over the data. This is nearly optimal, since simply storing the LIS itself requires $\Omega(k)$ space.

We also present lower bounds on the LIS problem in the streaming model. In the comparison model, Fredman [11] has proven that $n \log n - n \log \log n + \Theta(n)$ comparisons are necessary and sufficient to compute the LIS of an n-integer sequence, via a reduction from sorting. To the best of our knowledge, however, no lower bounds on LIS in the streaming model have been shown previously. As with many lower bounds on problems in the streaming model, our results are based upon the well-observed connection between the space required by a streaming algorithm and 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 $\Omega(k)$ 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.

For the LCS problem, we discuss a simple LIS-based algorithm requiring $O(n \log m)$ 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 sequence* against any number of test sequences, we can achieve the same space bound, independent of the number of test sequences. Our main results on LCS, however, are lower bounds. We prove that, if the two streams are general sequences, then we need $\Omega(n)$ space to ρ-approximate the LCS of two streams of length $\Omega(n)$ to within any factor ρ. If the given streams are n-element permutations, we prove that we need $\Omega(n/\rho^2)$ space to ρ-approximate the LCS.

2 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. We use a dynamic-programming style algorithm, maintaining the last element of the "best" increasing subsequence of length i seen so far, for each i less than or equal to the length of the LIS seen so far.

The algorithm presented here to calculate the length of the LIS was also discovered independently by Fredman [11] and by Bespamyatnikh and Segal [6] in a context other than the data streaming model; we include the algorithm here because our multipass algorithm to produce the LIS is an extension of it.

```
compute-LIS(X)
A[0] := −1
A[1] := ∞
k' := 0
WHILE there are elements left in the stream X
        Read in the next element x_i from X
        Find ℓ such that A[ℓ] ≤ x_i < A[ℓ + 1].
        Set A[ℓ + 1] := x_i
        IF ℓ + 1 > k'
            Set k' := k' + 1
            Set A[k' + 1] := ∞
Output k'
```

Figure 1: Pseudocode to compute the length of the LIS in a given stream X.

2.1 Computing the Length of an LIS

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

Our algorithm for computing the length of the LIS is based on maintaining $\langle \ell, j\rangle$-minimal subsequences for all $\ell \in \{1, \ldots, k'\}$ as we scan the stream, where k' is the length of the longest subsequence in the stream so far. Specifically, the streaming algorithm works as follows: we maintain an array $A[1 \ldots k']$, where, after we have scanned the first j elements of the stream, $A[\ell]$ will store $\mathsf{last}(\sigma)$ for an $\langle \ell, j\rangle$-MIS σ. The algorithm updates each $A[\ell]$ as new elements from the stream arrive, and increases k' as appropriate. See Figure 1 for the pseudocode.

Lemma 2.1 *After i iterations of the while loop in* $\mathsf{compute\text{-}LIS}()$*, we have*

$$A[\ell] = \begin{cases} \mathsf{last}(\rho) \text{ for } \rho \text{ an } \langle \ell, i\rangle\text{-MIS} & \text{if } \ell \leq \mathsf{LIS}(x_1, \ldots, x_i). \\ \infty \text{ or uninitialized} & \text{otherwise} \end{cases}$$

Proof. We proceed by induction on i, after strengthening the stated property by adding the following to the induction hypothesis:

$(*)$ $A[j] \leq A[j']$ for all $j < j'$ such that $A[j], A[j']$ are initialized.

For $i = 0$, the property is vacuously true. For the inductive case, assume the desired properties were maintained after we read in 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 subsequences of length $\ell + 1$ or higher might have a new smallest last element. That is, x_i is only going to affect values in A with indices $\ell + 1$ or higher.

On the other hand, note that x_i can only extend a previous increasing subsequence σ if σ ends with some element $\sigma_{|\sigma|} \leq x_i$. For all such subsequences, $\sigma_{|\sigma|} \leq x_i < A[\ell + 1]$. Hence by the induction hypothesis, σ is of length ℓ or shorter. This implies that the sequence $\sigma' = \sigma, x_i$ is of length at most $\ell + 1$. Thus 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. (We can extend the subsequence of length ℓ with last element $A[\ell]$). So it is necessary and sufficient that we update $A[\ell + 1]$. It is also clear that the new $A[j]$'s respect the ordering constraint. $\square$

Theorem 2.2 *We can decide whether the LIS of a given stream of integers from $\{1, \ldots, m\}$ has length at least a given number k, or compute the length k of the LIS of the given stream, with a one-pass streaming algorithm that uses $O(k \log m)$ space and has update time $O(\log k)$ or $O(\log \log m)$.*

Proof. By Lemma 2.1, the length is correctly computed by LIS. 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 keep k values in the range $\{1, \ldots, m\}$, i.e., $O(\log m)$ bits each. The only non-constant 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 binary search; alternatively, we can use a van Emde Boas queue [23] or y-fast trees [24] to support updates in $O(\log \log m)$ time. $\square$

2.2 Finding an LIS

The algorithm described in the previous section only computes the length of the LIS, but does not find such a sequence. We now present a multipass streaming algorithm that actually finds a longest increasing subsequence. Specifically, our algorithm finds the length-k LIS of a stream using $O(k^{1+\varepsilon} \log m)$ space in $\lceil \log(1 + 1/\varepsilon) \rceil$ passes over the data. We first explain the one-pass version of the algorithm, and then subsequently generalize it to multiple passes.

A one-pass algorithm. Consider an iteration in the decision algorithm in which we update, say, $A[\ell + 1]$ to x_i. In other words, we have $A[\ell] < x_i < A[\ell + 1]$. Then at this point, there is an increasing subsequence σ of length $\ell + 1$ whose last two elements are $A[\ell]$ and x_i, since x_i appears later in the stream than $A[j]$. Unfortunately, at some future time the value $A[j]$ may also be updated, and thus the old value is lost. (Thus, since the new $A[j]$ is later in the stream than x_i was, we can no longer reconstruct the last two elements of σ.)

The straightforward fix for this difficulty is, for each ℓ, to store the subsequence σ^ℓ of length ℓ that ends with $A[\ell]$. Thus, the algorithm maintains k sequences $\sigma^1, \ldots, \sigma^k$, taking a total of $O(k^2 \log m)$ space. When we update $A[\ell + 1] := x_i$, we reset $\sigma^{\ell+1} = \sigma^\ell, x_i$. This adds only a constant amount of extra running time per update, so the update time per element remains $O(\log k)$ or $O(\log \log m)$, and the space requirement is $O(k^2 \log m)$.

A two-pass algorithm. We now describe a two-pass algorithm that requires less space. The key modification is that during the first pass over the data, the algorithm only remembers part of each σ^ℓ, specifically every qth element (for a value of q to be specified below). For each ℓ, we maintain

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

where $\sigma^\ell_\ell = A[\ell]$, as before. The update rule for the first pass of the algorithm is then

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

where $\mathsf{all\text{-}but\text{-}last}(\widetilde{\sigma}^\ell)$ denotes the sequence $\widetilde{\sigma}^\ell$ with the last element of the sequence, $A[\ell] = \sigma^\ell_\ell$, omitted. Note that the space required for this entire pass is $O(k^2 \log m/q)$ when the length of the LIS is k.

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

$$\widetilde{\sigma}^k = \sigma_1^k, \sigma_{q+1}^k, \sigma_{2q+1}^k, \ldots, \sigma_{\lfloor\frac{k-2}{q}\rfloor q+1}^k, \sigma_k^k$$

where σ^k is a length-k LIS of the input. Write $\widetilde{\sigma}^k = z[1], z[2], \ldots, z[\lfloor(k-2)/q\rfloor], \sigma_k^k$.

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 ℓ. Notice that we can do this sequentially (for one ℓ at a time), since two consecutive τ subsequences do not overlap except at the endpoints. Thus each desired subsequence has length exactly $q+1$, and the total space required for the entire second pass is $O(q^2 \log m + k \log m)$.

Overall, the total space required by our algorithm is $O(\max(k^2 \log m/q, q^2 \log m) + k \log m)$. This is minimized at $q = k^{2/3}$, giving us a space bound of $O(k^{1+1/3} \log m)$ for two passes.

Generalizing to a p-pass algorithm. We can generalize this idea to a larger number of passes by computing the τ^ℓ subsequences recursively. As before, in the first pass the algorithm remembers only every qth element in the subsequence, and discards all stored subsequences except $\widetilde{\sigma}^k$. Then the algorithm uses $p-1$ passes to find the roughly k/q subsequences $\tau^1, \tau^2, \ldots, \tau^{\lfloor(k-2)/q\rfloor}$, where each τ^ℓ has length q.

Let $S(k,p)$ denote the space required by a p-pass algorithm to find a subsequence of length k. We then have the following recurrence: $S(k,p) = \max(O(k^2 \log m/q), S(q, p-1)) + O(k \log m)$. Solving the recurrence, we find that the space requirements are optimized at $q = k^{1-1/(2^p-1)}$, and where $S(k,p) = O(k^{1+1/(2^p-1)} \log m)$.

Theorem 2.3 *Fix any $\varepsilon > 0$. For a given k, we can find a length-k increasing subsequence of a given stream of integers from $\{1, \ldots, m\}$ with a $\lceil\log(1+1/\varepsilon)\rceil$-pass streaming algorithm that uses $O(k^{1+\varepsilon} \log m)$ space and has update time $O(\log k)$ or $O(\log\log m)$. We can find the longest increasing subsequence 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 $O(\frac{1}{\varepsilon} k^{1+\varepsilon} \log m)$.*

Proof. Given $\varepsilon > 0$, we choose $p = \lceil\log(1+1/\varepsilon)\rceil$. Then the p-pass algorithm described above uses space $O(k^{1+\varepsilon} \log m)$ to compute the LIS of the given stream.

If k is unknown, then we modify the algorithm described above slightly. Define a recursive sequence by $q_0 = 1$ and $q_{i+1} = q_i + q_i^{1-\varepsilon}$ for all $i \geq 0$. Then for the first pass only, change the update rule to the following:

$$\widetilde{\sigma}^{\ell+1} := \begin{cases} \widetilde{\sigma}^\ell, x_i & \text{if } \ell = q_i \text{ for some } i \\ \textsf{all-but-last}(\widetilde{\sigma}^\ell), x_i & \text{otherwise.} \end{cases}$$

So after the first pass, we have retained the sequence

$$\widetilde{\sigma}^k = \sigma_{q_0}^k, \sigma_{q_1}^k, \sigma_{q_2}^k, \ldots, \sigma_{q_t}^k, \sigma_k^k$$

where t is the largest index such that $q_t < k$.

By the recursion, the gap between adjacent indices $i, i+1 \leq t$ for elements of σ^k we have retained is $q_{i+1} - q_i = q_i^{1-\varepsilon} \leq k^{1-\varepsilon}$. In the standard algorithm where k is known in advance, we also have a gap of $k^{1-\varepsilon}$. So we can resume the standard algorithm from the second pass on, using the same time and space requirements.

Now, for the first pass of the algorithm, the update time is identical to the standard algorithm. However, the space used is $O(kt \log m)$. We now bound t. To this end, define $I_\phi = \{ i : 2^\phi \leq q_i < 2^{\phi+1} \}$. Let i_ϕ be the smallest index in I_ϕ. Then for all $j \geq 0$, we see $q_{i_\phi+j} \geq q_{i_\phi} + j q_{i_\phi}^{1-\varepsilon} \geq q_{i_\phi} + j 2^{\phi(1-\varepsilon)}$. Hence, $|I_\phi| \leq (2^{\phi+1} - 2^\phi)/2^{\phi(1-\varepsilon)} \leq 2^{\phi\varepsilon}$.

Let $I = \{i : q_i < k\}$. By definition, $t \leq |I|$. From the above, we have

$$|I| \leq \sum_{\phi=0}^{\lg k} |I_\phi| \leq \sum_{\phi=0}^{\lg k} 2^{\phi\varepsilon} = \frac{k^\varepsilon 2^\varepsilon - 1}{2^\varepsilon - 1} \leq \frac{k^\varepsilon 2^\varepsilon}{\varepsilon \ln 2}$$

where the last inequality follows since $e^x \geq 1 + x$ for all x. So the total space used in the first pass is $O(\frac{1}{\varepsilon} k^{1+\varepsilon} \log m)$, as we wanted. □

3 Lower Bounds for LIS

We now turn our attention to a lower bound on the space required for streaming algorithms solving the longest increasing subsequence 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 proof is based on a reduction from the *set disjointness* problem, which is known to have high communication complexity:

Definition 3.1 (Set Disjointness) *In the* SET-DISJOINTNESS *problem, there are two parties A and B who wish to solve the following problem. Party A holds an n-bit string s_A, and Party B holds another n-bit string s_B. They must decide whether there is at least one '1' in the bitwise-and s_A & s_B of s_A and s_B (i.e., decide if s_A and s_B both have a '1' in at least one position) while minimizing the number of bits communicated between the parties.*

We will say that s_A and s_B *intersect* for a "yes" instance of SET-DISJOINTNESS.

Lower bounds for the set disjointness problem are of fundamental importance, and have been studied extensively (e.g., [5, 18, 20]). The most recent results show that even in the randomized setting, SET-DISJOINTNESS requires a large number of bits of communication:

Proposition 3.2 ([5]) *Let $\delta \in (0, 1/4)$. Any randomized protocol solving the* SET-DISJOINTNESS *problem with probability at least $1-\delta$ requires at least $\frac{n}{4}(1-2\sqrt{\delta})$ bits of communication, even when 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 a stream of N elements. This reduction shows that—even if we allow randomization and some chance of error—deciding whether there is an increasing subsequence of length k requires $\Omega(k)$ space in the streaming model.

Suppose we are given an instance $\langle s_A, s_B \rangle$ of the SET-DISJOINTNESS problem, where $n := |s_A| = |s_B|$. We will construct a stream $\mathsf{lis\text{-}stream}(s_A, s_B)$ whose longest increasing subsequence has length $n+1$ if and only if $\langle s_A, s_B \rangle$ are non-disjoint. Further, the first half of $\mathsf{lis\text{-}stream}(s_A, s_B)$ depends only on s_A, while its second half depends only on s_B. With each index $i \in \{1, \ldots, n\}$, we associate the sequence $(n+1) \cdot (i-1) + 1, \ldots, (n+1) \cdot i$, divided into two parts: the first i integers form the sequence $\mathsf{A\text{-}part}(i) = (n+1) \cdot (i-1) + 1, (n+1) \cdot (i-1) + 2, \ldots, (n+1) \cdot (i-1) + i$ and the remaining $n-i+1$ integers form the sequence $\mathsf{B\text{-}part}(i) = (n+1) \cdot (i-1) + i + 1, (n+1) \cdot (i-1) + i + 2, \ldots, (n+1) \cdot i$.

Let $\mathsf{lis\text{-}stream\text{-}A}(s_A)$ be the sequence consisting of $\mathsf{A\text{-}part}(i)$ for every $i \in \{i : s_A(i) = 1\}$, in *decreasing* order of the index i. Similarly, let $\mathsf{lis\text{-}stream\text{-}B}(s_B)$ be the sequence consisting of $\mathsf{B\text{-}part}(i)$ for every $i \in \{i : s_B(i) = 1\}$, also listed in decreasing order of the index i. Clearly, $\mathsf{lis\text{-}stream\text{-}A}(s_A)$ (and $\mathsf{lis\text{-}stream\text{-}B}(s_B)$, respectively) only depends on s_A (s_B, respectively). Then we define the stream $\mathsf{lis\text{-}stream}(s_A, s_B)$ to be $\mathsf{lis\text{-}stream\text{-}A}(s_A)$ followed by $\mathsf{lis\text{-}stream\text{-}B}(s_B)$.

As an example (which we will return to throughout the paper), consider the 9-bit vectors $\mathsf{ex}_A = [0,1,0,1,1,0,0,0,0]$ and $\mathsf{ex}_B = [1,0,0,1,0,0,1,0,0]$. Then $n = 9$ and

$$\begin{aligned}\mathsf{lis\text{-}stream}(\mathsf{ex}_A, \mathsf{ex}_B) \;=\;& 41,42,43,44,45,\ 31,32,33,34,\ 11,12,\\ & \quad 68,69,70,\ 35,36,37,38,39,40,\ 2,3,4,5,6,7,8,9,10.\end{aligned}$$

Observe the increasing subsequence $31, 32, \ldots, 40$ of length $n+1 = 10$ in this stream.

Lemma 3.3 *The vectors s_A and s_B intersect if and only if $\mathsf{LIS}(\mathsf{lis\text{-}stream}(s_A, s_B))$ has length $n+1$.*

Proof. We prove the obvious direction first. If $s_A(i) = s_B(i) = 1$ for some particular i, then observe that $\mathsf{lis\text{-}stream}(s_A, s_B)$ contains the increasing subsequence $\mathsf{A\text{-}part}(i)\ \mathsf{B\text{-}part}(i) = (n+1)\cdot(i-1)+1, (n+1)\cdot(i-1)+2, \ldots, (n+1)\cdot i$ which contains $n+1$ increasing integers.

For the converse direction, we prove its contrapositive form. Suppose s_A and s_B do not intersect. Observe that whenever $i < j$ we have that (1) $\mathsf{A\text{-}part}(i)$ follows $\mathsf{A\text{-}part}(j)$ in $\mathsf{lis\text{-}stream\text{-}A}(s_A)$, 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 within $\mathsf{lis\text{-}stream\text{-}A}(s_A)$—or $\mathsf{lis\text{-}stream\text{-}B}(s_B)$, similarly—has length at most n, and can contain only the integers from $\mathsf{A\text{-}part}(i)$ for only a single i. Thus the only potential increasing subsequences of length $n+1$ must be subsequences of $\mathsf{A\text{-}part}(i)\ \mathsf{B\text{-}part}(j)$ 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 longest increasing subsequence in $\mathsf{lis\text{-}stream}(s_A, s_B)$ is of length at most $|\mathsf{A\text{-}part}(i)| + |\mathsf{B\text{-}part}(j)| = i + n - j + 1 \leq n$. □

We can improve the construction so that the resulting stream $\mathsf{LIS}(\mathsf{lis\text{-}stream}(s_A, s_B))$ is a *permutation*, i.e., a stream containing each of the numbers of $\{0, 1, \ldots, \ell\}$ exactly once. We will show that a suitable $\ell = \Theta(n^2)$ suffices. Our construction is an extension of the above. We modify $\mathsf{lis\text{-}stream\text{-}A}$ and $\mathsf{lis\text{-}stream\text{-}B}$ 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 so that only two of these elements can be part of an LIS:

- Let $U_A = \{x : x \in \mathsf{A\text{-}part}(i)$ for some i such that $s_A(i) = 0\}$. Then we define $\mathsf{pad\text{-}A}(s_A)$ to be the sequence consisting of integers in U_A listed in decreasing order, followed by 0. We define $\mathsf{lis\text{-}stream\text{-}perm\text{-}A}(S_A)$ to be $\mathsf{pad\text{-}A}(s_A)$ followed by $\mathsf{lis\text{-}stream\text{-}A}(s_A)$.

- Similarly, let $U_B = \{x : x \in \mathsf{B\text{-}part}(i)$ for some i such that $s_B(i) = 0\}$. Then we define $\mathsf{pad\text{-}B}(s_B)$ to be the sequence consisting of $(n+1)\cdot n + 1$, followed by the integers in U_B listed in decreasing order. We define $\mathsf{lis\text{-}stream\text{-}perm\text{-}B}(S_B)$ to be $\mathsf{lis\text{-}stream\text{-}B}(s_B)$ followed by $\mathsf{pad\text{-}B}(s_B)$.

Now define $\mathsf{lis\text{-}stream\text{-}perm}(s_A, s_B) := \mathsf{lis\text{-}stream\text{-}perm\text{-}A}(s_A)\ \mathsf{lis\text{-}stream\text{-}perm\text{-}B}(s_B)$. 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 our previous example, then,

$$\begin{aligned}\mathsf{lis\text{-}stream\text{-}perm}(\mathsf{ex}_A, \mathsf{ex}_B) \;=\;& 89,\cdots,81,78,\cdots,71,67,\cdots,61,56,\cdots,51,23,\cdots,21,1,\mathbf{0},\\ & 41,\cdots,45,\ 31,\cdots,34,\ 11,12,\\ & 68,\cdots,70,\ 35,\cdots,40,\ 2,\cdots,10,\\ & \mathbf{91},90,80,79,60,\cdots,57,50,\cdots,46,30,\cdots,24,20,\ldots,13.\end{aligned}$$

One can easily verify that $\textsf{lis-stream-perm}(s_A, s_B)$ is a permutation of the set $\{0, \ldots, (n+1)\cdot n+1\}$.

Lemma 3.4 *The vectors s_A and s_B intersect if and only $\textsf{LIS}(\textsf{lis-stream-perm}(s_A, s_B))$ has length at least $n+3$.*

Proof. Observe that the prefix of $\textsf{lis-stream-perm}(s_A, s_B)$ ending with the element 0 is a decreasing sequence, as is the suffix starting with the element $(n+1)\cdot n+1$. Thus any increasing subsequence of $\textsf{lis-stream-perm}(s_A, s_B)$ can contain at most one element from each of these segments. Thus the following sequence must be a longest increasing subsequence of $\textsf{lis-stream-perm}(s_A, s_B)$: first 0, then a longest increasing subsequence of $\textsf{lis-stream}(s_A, s_B)$, then $(n+1)\cdot n+1$. By Lemma 3.3, then, the length of the longest increasing subsequence of $\textsf{lis-stream-perm}(s_A, s_B)$ is $n+3$ if and only if s_A and s_B intersect. $\square$

Theorem 3.5 *For any length k and for any $N \geq k\cdot(k-1)+2$, any streaming algorithm which decides whether $\textsf{LIS}(\mathcal{S}) \geq k$ for a stream $\mathcal{S}$ which is a permutation of $\{1, \ldots, N\}$ with probability at least $3/4$ requires $\Omega(k)$ space.*

Proof. Suppose that an algorithm $\mathcal{A}(S)$ decides with probability at least $3/4$ whether stream S, where $|\mathcal{S}| = N$, 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 $|s_A| = k-1 = |s_B|$ with probability at least $3/4$ by calling $\mathcal{A}$. The stream we consider is

$$\mathcal{S} := \underbrace{N-1, N-2, \ldots, k\cdot(k-1)+2}_{\textsf{Extra Numbers}}, \textsf{lis-stream-perm}(s_A, s_B).$$

Note that, as in the proof of Lemma 3.4, the longest increasing subsequence of $\mathcal{S}$ has exactly the same length as the longest increasing subsequence of $\textsf{lis-stream-perm}(s_A, s_B)$ since the prepended elements of $\mathcal{S}$ are all larger than those in $\textsf{lis-stream-perm}(s_A, s_B)$, and are presented in descending order. Thus, by Lemma 3.4, the LIS of $\mathcal{S}$ has length k—and $\mathcal{A}(\mathcal{S})$ returns true with probability at least $3/4$—if and only if s_A and s_B do not intersect.

This immediately implies a lower bound on the space required by $\mathcal{A}$ by Proposition 3.2: to solve the instance $\langle s_A, s_B\rangle$ of the SET-DISJOINTNESS problem, Party A simulates the algorithm $\mathcal{A}$ on the stream $\textsf{Extra Numbers}, \textsf{lis-stream-perm-A}(s_A)$, then sends all stored information to Party B, who continues simulating $\mathcal{A}$ on the remainder of the stream $\mathcal{S}$. By Proposition 3.2, then, Party A must transmit at least $\Omega(k)$ bits in this protocol, and thus $\mathcal{A}$ must use $\Omega(k)$ space. $\square$

4 Longest Common Subsequence

In this section, we turn to the LCS problem. Recall that for LCS we are given two streams $\mathcal{S}_1$ and $\mathcal{S}_2$, consisting of n_1 and n_2 integers, respectively, drawn from the set $\{1, 2, \ldots, m\}$. Throughout this section, we consider the *adversarial* streaming model, in which elements from the two streams can be presented in any order of interleaving. Specifically, in the lower bounds that we construct in this section, the algorithm is given access to all of $\mathcal{S}_1$ before having access to any of $\mathcal{S}_2$.

First, as with all streaming problems, observe that there is a trivial streaming algorithm that solves LCS using $\Theta(n_1 \log m + n_2 \log m)$ space: we simply store both streams in their entirety, and then run a standard (non-streaming) LCS algorithm on the stored sequences. We can give another algorithmic upper bound for a version of LCS, based upon a simple connection between LIS and

LCS. Suppose that we are first given one *reference sequence* $\mathcal{R}$ and then given a large number of *test sequences* $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_q$; we want to compute the LCS of $\mathcal{R}$ and $\mathcal{S}_i$ for all $1 \leq i \leq q$. 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 2, where we interpret two elements x and y to be in ordered $x < y$ if x appears before y in $\mathcal{R}$. If these are n-element sequences, then this algorithm requires space $O(n \log m)$ total space—$O(n \log m)$ to store $\mathcal{R}$, and $O(k \log m) = O(n \log m)$ for the LIS computation. Note that this bound is independent of q.

In the remainder of this section, we present some lower bounds for LCS, again using the SET-DISJOINTNESS problem. We first show an easy lower bound when $\mathcal{S}_1$ and $\mathcal{S}_2$ are not necessarily permutations, and then show a more involved bound for exact or approximate computation of the LCS for permutations.

4.1 Lower Bound on Exact and Approximate LCS for General Sequences

It is straightforward to see that if we allow the streams $\mathcal{S}_1$ and $\mathcal{S}_2$ not to be permutations of each other, then the lower bound is trivial, even for approximation:

Theorem 4.1 *For any length N and any approximation ratio ρ, any streaming algorithm which ρ-approximates the LCS of two streams $\mathcal{S}_1, \mathcal{S}_2$ (in adversarial order) each of length N with probability at least $3/4$ requires $\Omega(N)$ space, even when the algorithm is presented all of $\mathcal{S}_1$ followed by all of $\mathcal{S}_2$.*

Proof. Let $\mathcal{S}$ be a sequence consisting of sequence $\mathcal{S}_1$ followed by sequence $\mathcal{S}_2$, and suppose that an algorithm $\mathcal{A}(\mathcal{S})$ decides with probability at least $3/4$ whether streams $\mathcal{S}_1$ and $\mathcal{S}_2$ contain a common subsequence of length 1. We show how to solve an instance $\langle s_A, s_B \rangle$ of SET-DISJOINTNESS with $|s_A| = 4N = |s_B|$, where s_A and s_B both contain exactly N ones, with probability at least $3/4$ by using $\mathcal{A}$.

Let stream $\mathcal{S}_1$ consist of all i such that $s_A(i) = 1$, and let $\mathcal{S}_2$ consist of all i such that $s_B(i) = 1$. Thus $\mathcal{S}_1$ and $\mathcal{S}_2$ have a common subsequence of length 1 if s_A and s_B have at least one element in common and of length 0 otherwise. Thus, if $\mathcal{A}(\mathcal{S})$ outputs the correct answer within any approximation ratio, it must distinguish between the 0 case and the length 1 case. This implies the desired lower bound, since we can solve the SET-DISJOINTNESS using $\mathcal{A}$. The first party simulates $\mathcal{A}$ on stream $\mathcal{S}_1$, then passes its state to the second party. The second party finishes simulating $\mathcal{A}$ on the rest of $\mathcal{S}$, namely on $\mathcal{S}_2$. By Proposition 3.2, this 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 new elements to each of the above streams. □

Although the above construction is for multiplicative approximation, a simple variation also shows that any data streaming algorithm solving this problem within additive α takes space at least $\Omega(N/\alpha)$; simply repeat each element in the streams 2α times.

4.2 Lower Bound on Exact LCS for Permutations

We now improve the construction to show a lower bound on the space required for an LCS algorithm even when the two streams $\mathcal{S}_1$ 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 each s_A and s_B, we construct two streams as follows:

- lcs-perm-A(s_A) consists of the sequence R_A followed by the sequence $\overline{R}_A$, where R_A contains $\{i : s_A(i) = 1\}$ in increasing order of i and $\overline{R}_A$ contains $\{i : s_A(i) = 0\}$ in decreasing order.

- lcs-perm-B(s_B) consists of the sequence R_B followed by the sequence $\overline{R}_B$, where R_B contains $\{i : s_B(i) = 1\}$ in increasing order and $\overline{R}_B$ contains $\{i : s_B(i) = 0\}$ in decreasing order.

Lemma 4.2 *The vectors s_A and s_B intersect if and only if* LCS(*lcs-perm-A*(s_A), *lcs-perm-B*(s_B)) *has length at least $n/2 + 2$.*

Proof. If s_A and s_B intersect, then we can construct a common subsequence of lcs-perm-A (s_A) and lcs-perm-B(s_B) as follows. First choose the common element from R_A and R_B. Since s_A and s_B intersect, the set $\{i : s_A(i) = s_B(i) = 0\}$ must contain at least $n/2 + 1$ elements, since there are exactly $n/4$ ones in each s_A and s_B. This implies a common subsequence of $\overline{R}_A$ and $\overline{R}_B$ of length $n/2 + 1$, and thus an overall common subsequence with total length $n/2 + 2$.

On the other hand, if A and B have no common element, then none of the elements in R_A can be matched up with R_B. Of course, some elements in R_A might be matched with elements in $\overline{R}_B$ (or vice versa), but R_A is in increasing order while $\overline{R}_B$ is in decreasing order, so at most one such element can be matched. Also $\overline{R}_A$ and $\overline{R}_B$ have exactly $n/2$ common elements, so $\overline{R}_A$ can at best be matched with at most $n/2$ elements in $\overline{R}_B$. Thus LCS(lcs-perm-A(s_A), lcs-perm-B(s_B)) can have length at most $n/2 + 1$. □

Theorem 4.3 *For any length k and for any $N \geq 2k - 4$, any streaming algorithm which decides whether* LCS($\mathcal{S}_1, \mathcal{S}_2$) $\geq k$ *for streams $\mathcal{S}_1, \mathcal{S}_2$ which are permutations of $\{1, \ldots, N\}$ with probability at least $3/4$ requires $\Omega(k)$ space.*

Proof. The theorem follows analogously to Theorem 4.1 when $N = 2k - 4$: deciding whether lcs-perm-A(s_A) and lcs-perm-B(s_B) have a common subsequence of length $N/2 + 2 = k$ requires $\Omega(k) = \Omega(N)$ space, by Lemma 4.2.

For larger N, we pad the streams, as in Theorem 3.5. Add the decreasing sequence $N, N - 1, N - 2, \ldots, 2k - 4 + 1$ to the beginning of lcs-perm-A(s_A) and add the increasing sequence $2k - 4 + 1, 2k - 4 + 2, \ldots, N$ to the end of lcs-perm-B(s_B). Then any common subsequences of these extended sequences are either (1) contained entirely in the unextended portions of lcs-perm-A(s_A) and lcs-perm-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. □

4.3 Lower Bound on Approximating LCS for Permutations

We now present lower bounds for the space required for approximation algorithms for LCS on permutations. Suppose that ρ is the desired approximation ratio. For each i, we will construct sequences ρ-approx-A(i,s_A) and ρ-approx-B(i,s_B) so that the two sequences have a common subsequence of length ρ^2 if $s_A(i) = s_B(i) = 1$, and so that the longest common subsequence has length at most ρ otherwise. For each $i \leq n$, both sequences are of length ρ^2, and consist of integers from $\{(i-1) \cdot \rho^2 + 1, (i-1) \cdot \rho^2 + 2, \ldots, (i-1) \cdot \rho^2 + \rho^2\}$. We define them as follows:

- For $s_A(i) = 1$, define ρ-approx-A(i,s_A) to be the increasing sequence $(i-1) \cdot \rho^2 + 1, (i-1) \cdot \rho^2 + 2, \ldots, (i-1) \cdot \rho^2 + \rho^2$. If $s_A(i) = 0$, then define ρ-approx-A(i,s_A) to be the decreasing sequence $(i-1) \cdot \rho^2 + \rho^2, (i-1) \cdot \rho^2 + \rho^2 - 1, \ldots, (i-1) \cdot \rho^2 + 1$.
- For $s_B(i) = 1$, define ρ-approx-B(i,s_B) to be the increasing sequence $(i-1) \cdot \rho^2 + 1, (i-1) \cdot \rho^2 + 2, \ldots, (i-1) \cdot \rho^2 + \rho^2$. When $s_B(i) = 0$, we use a more complicated ordering of the ρ^2 numbers. Specifically, we use what we call the *median sequence* σ of these ρ^2 numbers so that

the longest increasing subsequence and the longest decreasing subsequence of σ both have length exactly ρ. In this case, we define ρ-$\mathsf{approx\text{-}B}(i,s_B)$ to be the sequence

$$\begin{aligned}&(i-1)\cdot\rho^2+\rho,(i-1)\cdot\rho^2+\rho-1,\ldots,(i-1)\cdot\rho^2+1,\\&(i-1)\cdot\rho^2+2\rho,(i-1)\cdot\rho^2+2\rho-1,\ldots,(i-1)\cdot\rho^2+\rho+1,\\&\ldots\\&(i-1)\cdot\rho^2+\rho^2,(i-1)\cdot\rho^2+\rho^2-1,\ldots,(i-1)\cdot\rho^2+(\rho-1)\rho+1.\end{aligned}$$

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

$$\begin{aligned}&\mathsf{lcs}\text{-}\rho\text{-}\mathsf{approx\text{-}perm\text{-}A}(s_A)=\rho\text{-}\mathsf{approx\text{-}A}(1,s_A),\rho\text{-}\mathsf{approx\text{-}A}(2,s_A),\ldots,\rho\text{-}\mathsf{approx\text{-}A}(n,s_A)\quad\text{and}\\&\mathsf{lcs}\text{-}\rho\text{-}\mathsf{approx\text{-}perm\text{-}B}(s_B)=\rho\text{-}\mathsf{approx\text{-}B}(n,s_B),\rho\text{-}\mathsf{approx\text{-}B}(n-1,s_B),\ldots,\rho\text{-}\mathsf{approx\text{-}B}(1,s_B).\end{aligned}$$

Returning to our example from Section 3 where $n = 9$, we have

$$\begin{aligned}\mathsf{lcs\text{-}2\text{-}approx\text{-}perm\text{-}A}(\mathsf{ex}_A)&=4,3,2,1,\ 5,6,7,8,\ 12,11,10,9,\ 13,14,15,16,\ 17,18,19,20,\\&\quad\ 24,23,22,21,\ 28,27,26,25,\ 32,31,30,29,\ 36,35,34,33.\\\mathsf{lcs\text{-}2\text{-}approx\text{-}perm\text{-}B}(\mathsf{ex}_B)&=34,33,36,35,\ 30,29,32,31,\ 25,26,27,28,\ 22,21,24,23,\\&\quad\ 18,17,20,19,\ 13,14,15,16,\ 10,9,12,11,\ 6,5,8,7,\ 1,2,3,4.\end{aligned}$$

Lemma 4.4 *If s_A and s_B intersect, then $\mathsf{LCS}(\mathit{lcs}\text{-}\rho\text{-}\mathit{approx\text{-}perm\text{-}A}(s_A), \mathit{lcs}\text{-}\rho\text{-}\mathit{approx\text{-}perm\text{-}B}\ (s_B))$ has length at least ρ^2. If s_A and s_B do not intersect, then the length of the LCS is at most ρ.*

Proof. If s_A and s_B intersect, say with $s_A(i) = s_B(i) = 1$, then we see that ρ-$\mathsf{approx\text{-}A}(i,s_A) =$ ρ-$\mathsf{approx\text{-}B}(i,s_B)$. Hence the sequence $(i-1)\cdot\rho^2+1,\ldots,i\cdot\rho^2$ has length ρ^2 and is a subsequence of both lcs-ρ-$\mathsf{approx\text{-}perm\text{-}A}(s_A)$ and lcs-ρ-$\mathsf{approx\text{-}perm\text{-}B}(s_B)$. (In our example, $13, 14, 15, 16$ is such a subsequence.)

On the other hand, suppose s_A and s_B do not intersect. Recall that lcs-ρ-$\mathsf{approx\text{-}perm\text{-}A}$ (s_A) lists the ρ-$\mathsf{approx\text{-}A}(\cdot,s_A)$ in increasing order, while lcs-ρ-$\mathsf{approx\text{-}perm\text{-}B}(s_B)$ lists the ρ-$\mathsf{approx\text{-}B}(\cdot,s_B)$ in decreasing order. Thus any common subsequence can only have numbers that are a subsequence corresponding to exactly one index i. Since s_A and s_B do no intersect, we know that for any index i one of the three following cases holds:

1. $s_A(i) = 1, s_B(i) = 0$. Then ρ-$\mathsf{approx\text{-}A}(i,s_A)$ and ρ-$\mathsf{approx\text{-}B}(i,s_B)$ have a longest common subsequence of length ρ, since one is an increasing sequence while the other is a median sequence.

2. $s_A(i) = 0, s_B(i) = 1$. Then ρ-$\mathsf{approx\text{-}A}(i,s_A)$ and ρ-$\mathsf{approx\text{-}B}(i,s_B)$ have a longest common subsequence of length 1, since one is a decreasing sequence while the other is an increasing sequence.

3. $s_A(i) = 0, s_B(i) = 0$. Then ρ-$\mathsf{approx\text{-}A}(i,s_A)$ and ρ-$\mathsf{approx\text{-}B}(i,s_B)$ have a longest common subsequence of length ρ, since one is a decreasing sequence and the other is a median sequence.

Thus the LCS has length at most ρ when s_A and s_B do not intersect. □

Theorem 4.5 *For any approximation ratio ρ, and for any N, any streaming algorithm which decides 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 streams $\mathcal{S}_1, \mathcal{S}_2$ which are permutations of $\{1, \ldots, N\}$ with probability at least $3/4$ requires $\Omega(N/\rho^2)$ space.*

Proof. As in our previous lower-bound theorems, we can solve an instance of the SET-DISJOINTNESS problem with $|s_A| = N/\rho^2 = |s_B|$ as follows. By Lemma 4.4, deciding whether the constructed streams $\mathsf{lcs}\text{-}\rho\text{-}\mathsf{approx}\text{-}\mathsf{perm}\text{-}\mathsf{A}(s_A)$ and $\mathsf{lcs}\text{-}\rho\text{-}\mathsf{approx}\text{-}\mathsf{perm}\text{-}\mathsf{B}(s_B)$ have an LCS of length (i) at least ρ^2 or (ii) at most ρ corresponds to deciding whether s_A and s_B intersect. So a data stream algorithm $\mathcal{A}$ can be used to solve the SET-DISJOINTNESS problem. The first party simulates $\mathcal{A}$ on $\mathsf{lcs}\text{-}\rho\text{-}\mathsf{approx}\text{-}\mathsf{perm}\text{-}\mathsf{A}(s_A)$, then passes the state of the algorithm to the second party. The second party finishes the simulation of $\mathcal{A}$ on $\mathsf{lcs}\text{-}\rho\text{-}\mathsf{approx}\text{-}\mathsf{perm}\text{-}\mathsf{B}(s_B)$. Again, by Proposition 3.2, this implies that we need $\Omega(N/\rho^2)$ space for this LCS decision procedure. □

Corollary 4.6 *To ρ-approximate the LCS of N-element permutations, we need $\Omega(N/\rho^2)$ space.* □

5 Conclusion and Future Work

A classic theorem of Erdös and Szekeres 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 $\sqrt{n}$ [10]. 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 4 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.

Acknowledgements. We would like to thank D. Sivakumar for suggesting the problem to us, and for fruitful discussions. Thanks also to Graham Cormode for helpful discussions and comments.

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)*, 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] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 209–218, 2002.

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

[7] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*, 2002.

[8] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] M. R. Henzinger, P. Raghavan, and S. Rajagopalon. Computing on data streams. Technical Report 1998-011, Digital Equipment Corporation, Systems Research Center, May 1998.

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

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

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

[19] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. 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.

[20] A. A. Razborov. On the distributional complexity of disjointness. *Journal of Computer and System Sciences*, 28(2):260–269, 1984.

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

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

[23] 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.

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

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