Approximating the Longest Increasing Sequence and Distance from Sortedness in a Data Stream

Parikshit Gopalan[*]
Georgia Tech.
parik@cc.gatech.edu

T.S. Jayram
IBM Almaden
jayram@almaden.ibm.com

Robert Krauthgamer
IBM Almaden
robi@almaden.ibm.com

Ravi Kumar
Yahoo! Research
ravi-kuma@yahoo-inc.com

May 15, 2006

Abstract

We revisit the well-studied problem of estimating the sortedness of a data stream. We study the complementary problems of estimating the edit distance from sortedness (Ulam distance) and estimating the length of the longest increasing sequence (LIS). We present the first sub-linear space algorithms for these problems in the data stream model.

- We give a $O(\log^2 n)$ space, one-pass randomized algorithm that gives a $(4+\epsilon)$ approximation to the Ulam distance.
- We $O(\sqrt{n})$ space deterministic $(1+\epsilon)$ one-pass approximation algorithms for estimating the the length of the LIS and the Ulam distance.
- We show a tight lower bound of $\Omega(n)$ on the space required by any randomized algorithm to compute these quantities exactly. This improves an $\Omega(\sqrt{n})$ lower bound due to Vee *et al.* and shows that approximation is essential to get space-efficient algorithms.
- We conjecture a space lower bound of $\Omega(\sqrt{n})$ on any deterministic algorithm approximating the LIS. We are able to show such a bound for a restricted class of algorithms, which nevertheless captures all the algorithms described above.

Our algorithms and lower bounds use techniques from communication complexity and property testing.

[*]Work done in part while the author was at IBM Almaden

1 Introduction

In recent years, new computational models deigned for massive data sets, such as the data stream model of computation have been studied intensively across various branches of computer science. Much of this research focuses on revisiting basic algorithmic problems and designing streaming algorithms for them which are highly efficient with regard to storage space and update time. A problem that has received much attention in this model is estimating the sortedness of a data stream [EKK$^+$00, CMS01, AJKS02, GZ03].

It is well-known that one cannot hope for a sorting algorithm that is both space-efficient and makes only a few passes. Thus, it is natural to ask if one can at least estimate how far from sorted the data stream is. There are several scenarios where this problem arises. Consider for instance a huge index of web-pages sorted by a ranking function which is updated dynamically (for instance news stories). Since sorting is a relatively expensive operation, it is natural to look for an efficient procedure that will estimate the sortedness of the data on the fly; i.e. in a single pass and with very little storage space. On can run this relatively inexpensive procedure periodically, and based on the outcome decide when to sort the data.

There are several metrics possible for measuring the distance from sortedness (e.g. inversion distance, ℓ_1 distance) and there are efficient algorithms known for many of them. Ajtai *et al.* [AJKS02] and Gupta and Zane [GZ03] consider the problem of counting the number of inversions in a stream of numbers. Cormode *et al.* [CMS01] give data-stream algorithms for transposition distance and inversion distance. However, a natural metric which has proved harder for data stream algorithms and for other related computational models such as sketching and property-testing, is the edit-distance metric. This distance is also called the distance from monotonicity, or as Ulam distance from identity in the context of permutations. We will henceforth refer to it as the Ulam distance (see section 2 for precise definitions). This metric is particularly suited to measuring distance from sortedness, since it takes a global view of the input and is unaffected by the presence of a few out of place elements. But it is precisely this property that seems to make it harder to compute on a data stream. The problem of estimating edit distance from sortedness on a data-stream was posed as an open problem by Ajtai *et al.* [AJKS02] and Cormode *et al.* [CMS01].

This problem is closely related to the problem of computing the longest increasing subsequence (LIS) in a data stream, since the best way to sort the input is to identify a longest increasing subsequence and insert all the other elements into this subsequence. By abuse of notation, we will use LIS to denote both the longest increasing sequence and its length. The problem of estimating the LIS of an input sequence is again a well-studied problem with a long history (see the survey by Aldous and Diaconis [AD99]). There is a classical algorithm for this problem, known as Patience sorting, which can be viewed as a one-pass, $O(n)$ space, streaming algorithm. A natural question is whether once could have a more space-efficient algorithm for this problem. Vee *et al.* studied this problem and showed a space lower-bound of $\Omega(\sqrt{n})$ for exact computation. This leaves open the possibility of a streaming algorithm for exactly computing the LIS (and hence also the Ulam distance) using $o(n)$ space.

1.1 Our Results

1.1.1 Lower Bounds for Exact Computation

We show a lower bound of $\Omega(n)$ for any deterministic or randomized algorithm exactly computing the LIS of an n-element data stream. This also implies an $\Omega(n)$ lower bound for computing the Ulam distance. Thus for both these problems, we show that Patience-sorting is essentially optimal and getting a sub-linear space algorithm necessarily requires settling for approximation.

Theorem 1 *Given a string σ of length $2n$ over the alphabet $[4n]$, any randomized algorithm to decide whether* $\mathrm{LIS}(\pi)$ *is n or $n+1$ requires space $\Omega(n)$ even if multiple passes are allowed.*

To prove this bound, we first show that computing the AND of 2 bits reduces to computing the LIS on a string of length 2. We extend this reduction to show that set-disjointness reduces to computing the LIS. Further one can modify the reduction so that the lower bound holds even if the input is a permutation.

1.1.2 Deterministic Approximation Algorithm for LIS

We give an algorithm to compute a $(1-\epsilon)$ approximation to $\mathrm{LIS}(\sigma)$ using $O(\sqrt{\frac{n}{\epsilon}}\log m)$ space. One can view our algorithm as a version of Patience-Sorting with only bounded space available.

Theorem 2 *There is a one pass deterministic algorithm that computes a $(1-\epsilon)$ approximation to the LIS, using space $O(\sqrt{\frac{n}{\epsilon}}\log m)$ for any $\epsilon > 0$.*

To derive this algorithm, we first consider the following communication problem: Alice and Bob are respectively given strings σ_1 and σ_2. Their goal is to approximately compute the LIS of the string $\sigma = \sigma_1 \circ \sigma_2$ of length n. We give an efficient protocol for this problem in the one-way model where Alice sends a single message to Bob who then outputs the answer. The complexity of the protocol is independent of the length n of the input. This protocol generalizes naturally to the t-player case. Our algorithm may be viewed as a simulation of the protocol for $\sqrt{n}$ players using a data-stream algorithm. While this algorithm needs to know the length n of the data-stream in advance, we also present a modification that uses slightly more space, but does not need to know the length n.

One can derive an algorithm with similar parameters for Ulam distance.

1.1.3 Randomized Approximation Algorithm for Ulam Distance

For the problem of estimating the Ulam distance, we present a randomized algorithm that uses only $O(\log^2 n)$ space and gives a $(4+\epsilon)$ approximation.

Theorem 3 *There is a randomized algorithm that computes a $4+\epsilon$ approximation to the Ulam distance from identity with probability $1-\delta$ for arbitrarily small constant ϵ and δ. The space used as well as the update time per element are $O(\log^2 n)$ where n is the length of the data stream.*

Again, this algorithm does not need to known n in advance. If n is known to us, then we can get a high probability guarantee i.e δ can be taken to be inverse polynomial in n. The techniques used in our algorithm build on a body of work in property testing. Property testing algorithms for estimating the distance from monotonicity have been studied for sequences of numbers [EKK$^+$00], Boolean functions [GGL$^+$00] and for other more general domains [FLN$^+$02].

A first approach to computing the Ulam distance could be to relate it to the number of inversions in σ. However, it is well known that these quantities can be far apart. Ergun *et al.* [EKK$^+$00] show that a variation of this idea can be used to give a lower-bound on the Ulam distance. They consider the set of indices that are endpoints of an interval where the majority of elements are inverted w.r.t. the endpoint, and show that the cardinality of this set is a lower bound. Extending this observation, Ailon *et al.* [ACCL04] show that this quantity actually gives a factor-2 approximation to the Ulam distance.

We show that it actually suffices to consider the set of indices that are right end-points of such an interval (namely, where the majority of elements are inverted w.r.t. the endpoint), since the cardinality of this set provides a factor-4 approximation to the Ulam distance. This difference is in fact crucial to the design of a data-stream algorithm, since to decide whether a number is the right end-point of some such interval, we only need to know the numbers that have come before it in the data-stream. On the other hand, to decide if a number is the left end-point requires knowledge of the numbers that will come in the future, which is undesirable in the data-stream model.

We then devise a sampling scheme to test whether the index is the *right endpoint* of such an interval. Our sampling scheme is similar in spirit to the notion of Reservoir Sampling introduced by Vitter [Vit85]. The latter solves the following problem: given access to a set of inputs arriving in streaming fashion, we wish to produce, at any point in time, random elements from the set of all inputs seen so far, using a a small storage (memory). Our sampling scheme is more complicated, since we need produce samples from numerous different subsets of all the inputs seen so far, namely from the last k inputs seen for numerous different values of $k > 0$.

Finally, it is well-known that any property testing algorithm can be simulated by a streaming algorithm. Thus one can simulate the property testing algorithm of Ailon *et al.* [ACCL04] in streaming fashion. However, the space required is linear in the input size.

1.1.4 Lower Bounds for Approximating the LIS

We believe that the problem of approximating the LIS is harder than the problem of approximating the Ulam distance. As a first step in this direction, we conjecture that our approximation algorithm for the LIS which uses space $O(\sqrt{n})$ is optimal.

Conjecture 4 *Any deterministic one-pass streaming algorithm that computes a $(1+\epsilon)$ approximation to the LIS requires space $\Omega(\sqrt{n})$ for some constant $\epsilon > 0$.*

Part of the difficulty in proving this conjecture seems to arise from the following property of multi-player communication protocols for LIS: as the number of players increases, the *max communication complexity* of the protocols increases. Here max communication complexity refers to the maximum message size sent by any player. In contrast, for most problems including set-disjointness, the max communication reduces as the number of players increases. Indeed, the only other problem that we are aware of that shares this behavior is the problem of approximately counting the number of 1's in a data-stream of length n. There is a lower bound of $\Omega(\log n)$ for deterministic streaming algorithms for this problem due to Ajtai [Ajt02], however the techniques used are not from communication complexity.

We are able to establish this conjecture for limited classes of algorithms, which store some subset of the sequence seen so far in their memory. This class captures all the of the algorithms mentioned in this paper. Our result builds on a lower bound technique for streaming algorithms due to Ajtai [Ajt02].

2 Preliminaries

Given a sequence σ of length n over an alphabet $\{1, \cdots, m\}$, an increasing sequence in σ is a subsequence $(i_1, \cdots, i_k)$ so that

$$i_1 < i_2 \cdots < i_k, \quad \sigma(i_1) \le \sigma(i_2) \cdots \le \sigma(i_k)$$

Let $\mathrm{LIS}(\sigma)$ denote the length of the longest increasing sequence in σ.

A sequence σ is monotone if $\sigma(1) \leq \sigma(2) \cdots \leq \sigma(n)$. We consider edit operations on a sequence where a single edit operation involves deleting a single element of the sequence and inserting it in a new place. For a sequence σ, the distance from monotonicity is the minimum number of edit operations that must be performed to transform σ into a monotone sequence σ' and is denoted by $\mathrm{Ed}(\sigma)$. It holds that

$$\mathrm{LIS}(\sigma) + \mathrm{Ed}(\sigma) = n \tag{1}$$

since the best way to make σ monotone is to identify a longest increasing subsequence and insert all the other elements into this subsequence.

The Ulam distance between two permutations is the number of edit operations needed to convert one permutation to the other, where a single edit operation involves deleting a character and inserting it in a new place. Note that $\mathrm{Ed}(\pi)$ is just the Ulam distance between π and the identity permutation. It measures of how far from sorted the permutation π is.

We use i, j, k to denote lengths of sequences and a, b, c to denote letters of the alphabet.

Definition 1 *Given a sequence σ, for $i \in \{1, \cdots, n\}$ let $P_\sigma(i)$ be the smallest letter a such that there is an increasing sequence of length i in σ ending at a.*

For $i \geq \mathrm{LIS}(\sigma)$ there is no increasing sequence in σ of length i, so we will set $P_\sigma(i) = \infty$. When the string σ is clear from the context, we will just use $P(i)$. Note that $P_\sigma(i)$ is an array of letters and that it is monotone.

Definition 2 *Given a sequence σ, for $a \in \{1, \cdots, m\}$ let $L_\sigma(a)$ be the largest i such that there is an increasing sequence of length i in σ ending at or before a.*

The array L_σ is an array of lengths, indexed by letters of the alphabet. Since our definition allows for sequences that and at or before a, it follows that L_σ is also monotone. Also $L_\sigma(m) = \mathrm{LIS}(\sigma)$.

3 Lower Bounds for Exact Algorithms

We consider the communication complexity of the following problem: Alice is given the first half of σ, Bob is given the second half and they wish to compute $\mathrm{LIS}(\sigma)$. We give a reduction from the problem of computing the AND of two bits to computing the LIS when $n = 2$ and $m = 4$. Assume Alice holds a bit x, Bob holds a bit y and they want to compute $x \wedge y$. If $x = 0$ then $\sigma(1) = 4$ else $\sigma(1) = 2$. If $y = 0$, then $\sigma(2) = 1$ else $\sigma(2) = 3$. It is easy to verify that $\mathrm{LIS}(\sigma) = 2$ if $x \wedge y = 1$ and that $\mathrm{LIS}(\sigma) = 1$ otherwise.

We can extend this to get a reduction from Set Disjointness to computing $\mathrm{LIS}(\sigma)$. Assume that Alice and Bob have strings $x = x_1, \cdots, x_n$ and $y = y_1, \cdots, y_n$ respectively which represent the incidence vectors of sets X, Y on $[n]$. Alice uses x to compute the string $\sigma(i)$ for $1 \leq i \leq n$ using

$$\sigma(i) = 4(i-1) + 4 \quad \text{if} \quad x_i = 0$$
$$\sigma(i) = 4(i-1) + 2 \quad \text{if} \quad x_i = 1$$

Bob uses y to compute the string $\sigma(n+i)$ for $1 \leq i \leq n$ using

$$\sigma(n+i) = 4(i-1) + 1 \quad \text{if} \quad y_i = 0$$
$$\sigma(n+i) = 4(i-1) + 3 \quad \text{if} \quad y_i = 1$$

Let $\sigma = \sigma(1) \cdots \sigma(2n)$.

Proposition 5 *If the sets X and Y intersect then* $\mathrm{LIS}(\sigma) = n+1$. *If the sets are disjoint then* $\mathrm{LIS}(\sigma) = n$.

Proof: Assume that X and Y are disjoint. We claim that any increasing sequence can contain at most one element from each interval $[4(i-1)+1, 4(i-1)+4]$ for $1 \leq i \leq n$. This is because the string σ contains precisely two elements in this interval $\sigma(i)$ and $\sigma(n+i)$ and they are in increasing order iff $x_i \wedge y_i = 1$. Hence when $X \cap Y$ is empty, only one of them can occur in any increasing sequence and $\mathrm{LIS}(\sigma) \leq n$. Equality holds since $\sigma(1), \cdots, \sigma(n)$ is an increasing sequence.

On the other hand, assume that $i \in X \cap Y$. The following is an increasing subsequence of σ:

$$\sigma(1), \cdots, \sigma(i), \sigma(n+i), \cdots, \sigma(2n)$$

It is easy to check that in fact $\mathrm{LIS}(\sigma) = n+1$. $\square$

Theorem 1 follows from known lower bounds on the communication complexity of set-disjointness [BYJKS02]. Note that the sequences use in the proof above are not permutations. However one can get a reduction from the problem of computing the AND of two bits to computing the LIS when $n = m = 8$. Assume Alice holds a bit x, Bob holds a bit y and they want to compute $x \wedge y$. If $x = 0$ then $\sigma(1) = 4$ else $\sigma(1) = 2$. If $y = 0$, then $\sigma(2) = 1$ else $\sigma(2) = 3$. It is easy to verify that $\mathrm{LIS}(\sigma) = 2$ if $x \wedge y = 1$ and that $\mathrm{LIS}(\sigma) = 1$ otherwise.

4 Deterministic Approximation Algorithms

4.1 Patience Sorting

Patience Sort is a dynamic program to exactly compute $\mathrm{LIS}(\sigma)$ and $\mathrm{Ed}(\sigma)$. It can be viewed as a one-pass streaming algorithm.

```
Patience Sort
Set $P(i) = \infty$ for all $i$.
For $j \in [1, n]$
    Let $\sigma(j) = a$.
    Find the largest $i$ so that $P(i) \leq a$.
    Set $P(i+1) = a$.
Output the largest $i$ so that $P(i) \neq \infty$.
```

It is easy to see that Patience Sort computes $P(i)$ correctly. The space used is $\mathrm{LIS}(\sigma) \log m \leq n \log m$. Another approach to computing $\mathrm{LIS}(\sigma)$ would be to keep track of $L(a)$ for $a \in \{1, \cdots, m\}$. The space needed for this would be $m \log n$.

4.2 A Two Player Protocol to Approximate LIS

Consider the following problem in two-player communication complexity. Alice is given a string σ_1, Bob is given a string σ_2 and they wish to compute a $(1-\epsilon)$ approximation to $\mathrm{LIS}(\sigma)$ where $\sigma = \sigma_1 \circ \sigma_2$. We will give a one-way protocol for this problem which uses $\epsilon^{-1} \log m + \log n$ bits of communication.

```
Two player Protocol for LIS
1.  Alice runs Patience Sort on $\sigma_1$.  She computes $P_{\sigma_1}$ and $k_1 = \mathrm{LIS}(\sigma_1)$.
2.  She sends Bob $P_{\sigma_1}(i)$ for all $i$ multiples of $\epsilon k$.
3.  Bob computes the best extension of these sequences by $\sigma_2$.
4.  He outputs $k$ which is the length of the longest sequence.
```

Lemma 6 *The value k output by Bob satisfies $k > (1-\epsilon)\,\mathrm{LIS}(\sigma)$.*

Proof: Assume the LIS is of the form $\pi_1 \circ \pi_2$ where π_1 is a substring of σ_1 and π_2 is a substring of σ_2. Assume that $|\pi_1| = \ell_1$ and $|\pi_2| = \ell_2$ so that $\mathrm{LIS}(\sigma) = \ell_1 + \ell_2$.

Let $\pi_(\ell_1) = a$ and $\pi_2(1) = b$ so that $a \leq b$. Choose ℓ_1' to be a multiple of ϵk such that

$$\ell_1 - \epsilon k \ \leq \ \ell_1' \ \leq \ \ell_1$$

Since π_1 is an increasing sequence,

$$\pi_1(\ell_1') \ \leq \ a$$

Let $P_{\sigma_1}(\ell_1') = a'$. From the definition of P_{σ_1},

$$a' \ \leq \ \pi_1(\ell_1')$$

Hence we have

$$a' \ \leq \ \pi_1(\ell_1') \ \leq \ a \ \leq \ b$$

Thus Alice's message tells Bob that σ_1 contains an increasing sequence of length ℓ_1' ending at $a' \leq a$. Bob can extend this sequence by π_2 to get an increasing sequence of length $\ell_1' + \ell_2$. Thus

$$k \ \geq \ \ell_1' + \ell_2 \ \geq \ell_1 - \epsilon k_1 + \ell_2 \ \geq \ \mathrm{LIS}(\sigma) - \epsilon k_1 \ \geq (1-\epsilon)\,\mathrm{LIS}(\sigma)$$

The last inequality holds since $\mathrm{LIS}(\sigma) \geq \mathrm{LIS}(\sigma_1)$. $\square$

In fact, a stronger statement is true. Assume that at the end of the protocol we ask Bob to estimate the value of $L(a)$ for every $a \in \{1, \cdots, m\}$. For each a Bob outputs $L'(a) = i$ where $i \in S$ is the largest index so that $P'(i) \leq a$. Then for every a, we have

$$L(a) - \epsilon k_1 \ \leq \ L'(a) \ \leq L(a)$$

The proof is identical to that of Lemma 6.

4.3 Streaming Algorithm for Approximate LIS

We give an algorithm to compute a $(1-\epsilon)$ approximation to $\mathrm{LIS}(\sigma)$ using $O(\sqrt{\frac{n}{\epsilon}}\log m)$ space. One can view this algorithm as a version of Patience Sort with only bounded space available. We compute values $P'(i)$ which are meant to approximate $P(i)$. However unlike in Patience Sort, we want to ensure that the set of indices i for which $P'(i)$ is stored is never more than $2\sqrt{\frac{n}{\epsilon}}$ in size. Thus the algorithm proceeds similar to Patience Sort, except if the number of stored values exceeds this bound. Then we do a cleanup operation, where we only retain the values of P' for $\sqrt{\frac{n}{\epsilon}}$ evenly spaced indices i.

```
Streaming LIS
Set $S = \{0\}, P'(0) = 0$.
For $j \in 1, \cdots, n$
    Let $\sigma(j) = a$.
    Find the largest $i \in S$ so that $P'(i) \le a$.  Set $P'(i+1) = a$.
    If $i + 1 \notin S$, add it to $S$.
    If $|S| > 2\sqrt{\frac{n}{\epsilon}}$          (Cleanup)
        Let $k = \max\{i \in S\}$.
        Set $S = \{\sqrt{\frac{\epsilon}{n}}k, 2\sqrt{\frac{\epsilon}{n}}k, \cdots, k\}$.
        Store $P'(i)$ only for $i \in S$.
Output $k = \max\{i \in S\}$.
```

Note that we only store P' for some set of indices S. However we think of P' as a function on the interval $[1, k]$ where k is the largest value in S using the following definition: For $i \notin S$ we set $P'(i) = P'(j)$ for the smallest $j > i$ which lies in S. The motivation for this definition is that $P'(i)$ is the smallest letter a so that the algorithm detects an increasing sequence of length i ending at a. If $P'(j) = a$ and $j > i$ then certainly we have $P'(i) \le a$.

Proposition 7 *The algorithm performs less than $\sqrt{\epsilon n}$ cleanup operations.*

Proof: Between two consecutive cleanup operations, the set S grows by $\sqrt{\frac{n}{\epsilon}}$, hence the value of j increases by at least this amount. Since the stream is of length n, in total there are no more than $\sqrt{\epsilon n}$ cleanup operations. □

Based on when the cleanups occur, we can break the string σ as $\sigma_1 \circ \sigma_2 \cdots \sigma_{\sqrt{\epsilon n}}$ (the last few might be empty). We bound the performance of the algorithm by comparing it to Patience Sort. This is done by comparing the values of P' and P prior to every cleanup operation. Let P'_t denote the function P' after processing $\sigma_1 \circ \cdots \circ \sigma_t$ and prior to the t^{th} cleanup. Let us denote $P_{\sigma_1 \circ \cdots \sigma_t}$ by P_t. Let k_t be the largest value in S at this point. We have $k_1 \le k_2 \cdots \le k_t$ since the algorithm never discards the largest element in S. Also we have $k_t \le \mathrm{LIS}(\sigma)$ since our algorithm detects an increasing sequence of length k_t.

The intuition for the analysis is that at every step, the algorithm maintains an additive approximation to $L(a)$. Each new cleanup operation causes the error to increase, but it can be bounded by $t\sqrt{\frac{\epsilon}{n}}k_{t-1}$ after t cleanup operations.

Lemma 8 *For $1 \le t \le \sqrt{\epsilon n}$,*

$$P'_t\left(i - (t-1)\sqrt{\frac{\epsilon}{n}}k_{t-1}\right) \;\le\; P_t(i)$$

Proof: The base case when $t = 1$ is trivial, since in this case $P'_1(i) = P_1(i)$. Assume by induction that the claim holds for $t - 1$.

We now consider what happens to P and P' after processing the string σ_t. After the $(t-1)^{st}$ cleanup, the memory contains the value of P'_{t-1} for all multiples of $\sqrt{\frac{\epsilon}{n}}k_{t-1}$. P'_t is obtained by computing the best possible extensions of these sequences using σ_t.

Let $P_t(i) = c$. Consider the increasing sequence of length i ending at c. Split this sequence into $\pi_1 \circ \pi_2$ where π_1 lies in $\sigma_1 \circ \cdots \sigma_{t-1}$ and π_2 lies in σ_t. Assume that they have length ℓ_1 and ℓ_2 respectively so that $\ell_1 + \ell_2 = i$. Let b be the first letter of π_2 and a the last letter of π_1 so that

$a \leq b \leq c$. We may assume that $P_{t-1}(\ell_1) = a$, since if there is an increasing sequence of length ℓ_1 that ends earlier than a, we could use it to replace π_1. Applying the induction hypothesis to $P'_{t-1}(\ell_1)$,

$$P'_{t-1}\left(\ell_1 - (t-2)\sqrt{\frac{\epsilon}{n}}k_{t-2}\right) \;\leq\; P_{t-1}(\ell_1) \;= a \tag{2}$$

We can find ℓ'_1 which is a multiple of $\sqrt{\frac{\epsilon}{n}}k_{t-1}$ satisfying

$$\ell_1 - (t-2)\sqrt{\frac{\epsilon}{n}}k_{t-2} - \sqrt{\frac{\epsilon}{n}}k_{t-1} \;\leq\; \ell'_1 \;\leq\; \ell_1 - (t-2)\sqrt{\frac{\epsilon}{n}}k_{t-2} \tag{3}$$

Since $k_{t-1} \geq k_{t-2}$, we can lower bound ℓ'_1 by

$$\ell'_1 \;\geq\; \ell_1 - (t-1)\sqrt{\frac{\epsilon}{n}}k_{t-1} \tag{4}$$

Since P'_{t-1} is a monotone function on $i \in [1, k_{t-1}]$, from equations 2 and 3 we get

$$P'_{t-1}(\ell'_1) \;\leq\; P'_{t-1}(\ell_1 - (t-2)\sqrt{\frac{\epsilon}{n}}k_{t-2}) \;\leq\; P_{t-1}(\ell_1) = a \tag{5}$$

Since ℓ'_1 is a multiple of $\sqrt{\frac{\epsilon}{n}}k_{t-1}$, this value is stored in the memory even after the $(t-1)^{st}$ cleanup. We can extend this sequence using π_2 and get an increasing sequence ending at a of length

$$\ell'_1 + \ell_2 \;\geq\; \ell_1 - (t-1)\sqrt{\frac{\epsilon}{n}}k_{t-1} + \ell_2 \;\geq\; i - (t-1)\sqrt{\frac{\epsilon}{n}}k_{t-1}$$

Hence we have

$$P'_t\left(i - (t-1)\sqrt{\frac{\epsilon}{n}}k_{t-1}\right) \;\leq\; a \;= P_t(i)$$

which completes the induction. □

Theorem 9 *There is a one pass deterministic algorithm that computes a $(1-\epsilon)$ approximation to the LIS, using space $O(\sqrt{\frac{n}{\epsilon}}\log m)$ for any $\epsilon > 0$.*

Proof: Assume that LIS$(\sigma) = k$ and the LIS ends at a so that $P(k) = a$. Assume that the total number of cleanups is $t \leq \sqrt{\epsilon n}$. Applying Lemma 8, we get

$$P(k - t\sqrt{\frac{\epsilon}{n}}k_{t-1}) \leq a$$

Hence the algorithm detects an increasing sequence of length k_t where

$$k_t \;\geq\; k - t\sqrt{\frac{\epsilon}{n}}k_{t-1} \;\geq\; k - \epsilon k_{t-1} \;\geq\; (1-\epsilon)k$$

where the last inequality uses $k_{t-1} \leq k$. □

4.4 Another Streaming Algorithm

The Algorithm in the last section needs to know the length n of the data stream in advance. In this section, we give an algorithm that does not need to know n. However the space used becomes $O(n^{1/2+\delta})$ where we can take any constant $\delta > 0$. One can view the previous algorithm as a simulation of a $\sqrt{\epsilon n}$-player one-way protocol which generalizes the two-player protocol in the last section. Each players read $\sqrt{\frac{\epsilon}{n}}$ elements from the input an sends a sketch of the data stream so far to the next player. Each player introduces an additive error of $\sqrt{\frac{\epsilon}{n}}\,\mathrm{LIS}(\sigma)$. If the size of the input is not known in advance, then a reasonable scheme would be to have the t^{th} player read t elements of the data stream. In this way, each player reads roughly $\sqrt{n}$ elements. The problem with this scheme is that the additive error grows as $\sum_t \frac{\epsilon}{t}\,\mathrm{LIS}(\sigma)$ which is unbounded as t goes to infinity. To overcome this, we ensure that the additive error introduced by the th player is no more than $\frac{\epsilon}{t^{1+\delta}}\,\mathrm{LIS}(\sigma)$. Since $\sum_t t^{-(1+\delta)}$ converges, the total error is bounded by a constant fraction of $\mathrm{LIS}(\sigma)$.

```
Streaming LIS
Set $S = \{0\}, P'(0) = 0, t = 1$.
For $j \in 1, \cdots, n$
    Let $\sigma(j) = a$.
    Find the largest $i \in S$ so that $P'(i) \le a$.  Set $P'(i+1) = a$.
    If $i + 1 \notin S$, add it to $S$.
    If $|S| > 2\frac{t^{1+\delta}}{\epsilon}$          $(Cleanup)$
        Let $k = \max\{i \in S\}$.
        Set $S = \{\frac{\epsilon}{t^{1+\delta}}k, 2\frac{\epsilon}{t^{1+\delta}}k, \cdots, k\}$.
        Store $P'(i)$ only for $i \in S$.
        Set $t = t + 1$.
Output $k = \max\{i \in S\}$.
```

Since the gap between two cleanups is at least $t^{1+\delta}$, on an input of length n we have $t = O(n^{\frac{1}{2+\delta}})$. Hence the total space used can be bounded by $O(\epsilon^{-1} n^{\frac{1+\delta}{2+\delta}} \log m)$. The correctness is based on the following Lemma whose proof is similar to that of Lemma 8. Again we break the string σ into $\sigma_1 \circ \sigma_2 \cdots$. We define P_t, k_t and P'_t as before.

Lemma 10 *For $1 \le t \le n^{\frac{1}{2+\delta}}$,*

$$P'_t\left(i - \epsilon k_{t-1} \sum_{j=1}^{t-1} \frac{1}{j^{1+\delta}}\right) \;\le\; P_t(i)$$

Theorem 11 *There is a one pass deterministic algorithm that computes a $(1-\epsilon')$ approximation to the LIS, using space $O(n^{\frac{1+\delta}{2+\delta}} \log m)$ for any $\epsilon' > 0$ which does not need to know the length of the stream in advance.*

Proof: Assume that $\mathrm{LIS}(\sigma) = k$. By Lemma 10, the algorithm detects an increasing sequence of length k_t where

$$k_t \;\ge\; k - \epsilon k_{t-1} \sum_{j=1}^{t-1} \frac{1}{j^{1+\delta}}$$

The sum $\sum_{j=1}^{t-1} \frac{1}{j^{1+\delta}}$ is bounded above by an absolute constant depending only on δ. So by taking ϵ sufficiently small, we can get

$$k_t \;\geq\; k - \epsilon' k_{t-1} \;\geq\; (1-\epsilon')k$$

□

5 Estimating Ulam Distance

5.1 A characterization via inversions

A pair of indices (i,j) is said to be inverted in σ if $i > j$ but $\sigma(i) < \sigma(j)$. For a given index i let $\mathrm{Inv}(i)$ denote the set of indices j that are inverted with respect to i. We define a set R which consists of all indices i that are the right end-points of a interval where the majority of elements are inverted with respect to i. Formally

$$R = \{i \in [n] \mid \exists\ j\ s.t. \text{ majority of indices in [j, i-1] lie in } \mathrm{Inv}(i)\}$$

We shall interpret majority as being a strict majority. In particular if $(i-1,i)$ is an inversion then $i \in R$. More generally for $\delta \leq 1/2$ we define R_δ to consist of all indices i that are the right end-points of a interval where a δ fraction of elements are inverted with respect to i.

$$R_\delta = \{i \in [n] \mid \exists\ j\ s.t. \text{ more than } \delta \text{ fraction of indices in [j, i-1] lie in } \mathrm{Inv}(i)\}$$

Lemma 12 *The following bound holds for all $\delta \leq 1/2$:*

$$\frac{\mathrm{Ed}(\sigma)}{2} \;\leq\; |R| \;\leq\; |R_\delta| \;\leq\; \frac{\mathrm{Ed}(\sigma)}{\delta}$$

We first show that $\mathrm{Ed}(\sigma) \leq 2|R|$. We give an algorithm that deletes at most $2|R|$ indices and returns an increasing subsequence of σ. Assume w.l.o.g that $\sigma(n+1) = m$ so $n+1 \notin R$. The algorithm begins with $i = n+1$ and scans the string from right to left. If $i-1 \notin R$ then it moves to $i-1$. Else it skips to the first $j < i$ that is not in R or in $\mathrm{Inv}(i)$. It deletes all the indices in $[j+1, i-1]$.

Proposition 13 *The algorithm deletes at most $2|R|$ indices and returns an increasing sequence.*

Proof: We claim that a majority of the indices that are deleted at any step lie in R. To see this, let $i \notin R$ and let j be the largest index such that $j < i$ and j does not belong to either of $\mathrm{Inv}(i)$ or R. Every element in $[j+1, i-1]$ lies in $\mathrm{Inv}(i)$ or R. But since $i \notin R$, at least half the indices from $[j+1, i-1]$ do not lie in $\mathrm{Inv}(i)$ hence they lie in R.

The algorithm returns a subsequence $(i_1, \cdots, i_k)$ so that $(i_{\ell-1}, i_\ell)$ is not an inversion. Thus consecutive elements are in the right order, so the entire sequence is monotone. □

The inclusion $R \subseteq R_\delta$ follows from the definition hence $|R| \leq |R_\delta|$. To prove the upper bound, fix a set $D \subseteq [n]$ of indices of size $\mathrm{Ed}(\sigma)$ so that deleting D leaves a monotone sequence. Note that the set D may not unique. Define S_δ to consist of all indices $i \in D^c$ that are the right end-points of a interval where a δ fraction of elements lie in D.

$$S_\delta = \{i \in [n] | i \in D^c,\ \exists\ j\ s.t. \text{ more than } \delta \text{ fraction of indices in [j, i-1] lie in } D\}$$

We say that such a j is a witness for the membership of i in S_δ. We give an algorithm than scans left to right and computes the set S_δ. Start with the smallest index $j \in D$. Find the smallest index $k > j$ so that at most δ fraction of $[j, k-1]$ lies in D. Add the indices in $[j, k-1] \cap D^c$ to S_δ. Let ℓ be the smallest index greater than k that lies in D. Set $j = \ell$ and repeat.

Proposition 14 *For any $\delta \leq 1/2$,*

$$|S_\delta| \leq \mathrm{Ed}(\sigma)\left(\frac{1}{\delta} - 1\right)$$

Proof: Assuming that the set output by the algorithm is in fact S_δ, it is clear that

$$|D| + |S_\delta| \leq \frac{1}{\delta}|D|$$

The bound follows since $|D| = \mathrm{Ed}(\sigma)$. We need to show that S_δ is computed correctly, this is simple but tedious case analysis.

We show that the algorithm correctly computes the set $S_\delta \cap [1, \ell-1]$. It is easy to see that $[1, j]$ do not lie in S_δ. Fix an index $i \in [j, k-1] \cap D^c$. Since $i < k$ so by the choice of k, at least a δ fraction of $[j, i-1]$ lies in D which shows $i \in S_\delta$.

To show $k \notin S_\delta$, let j' be a potential witness. If $j' \in [j, k-1]$, partition the interval $[j, k-1]$ into $[j, j'-1]$ and $[j', k-1]$. By the choice of k, more than δ fraction of $[j, j'-1]$ lies in D. If the same holds for $[j', k-1]$, then it also holds for $[j, k-1]$ but this contradicts the choice of k. So j' cannot be a witness. On the other hand, if $j' < i$, then the ratio of elements from D only decreases, so $k \notin S_\delta$. Similarly, we can show that any $i \in [k, \ell-1]$ does not lie in S_δ. Hence, the algorithm correctly identifies the set $S_\delta \cap [1, \ell-1]$.

Also, if $i > \ell$ lies in S_δ, there is always a witness $j \geq \ell$. This is because if $j < \ell$ is a witness, we can partition the interval $[j, i-1]$ into $[j, \ell-1]$ and $[\ell, i-1]$. The first interval has at most a δ fraction from D since $\ell - 1 \notin S_\delta$. Hence $[\ell, i-1]$ contains more than δ fraction from D so ℓ serves as a witness. □

Proposition 15 *For any $\delta \leq 1/2$,*

$$|R_\delta| \leq \frac{\mathrm{Ed}(\sigma)}{\delta}$$

Proof: We can partition R into $R \cap D$ and $R \cap D^c$. Clearly

$$|R \cap D| \ \leq \ |D| \ = \ \mathrm{Ed}(\sigma)$$

We now bound the size of $R \cap D^c$. Note that the set D^c forms an increasing sequence. Thus if $i, j \in D^c$ then they are not inverted. Hence for any $i \in D^c$, we have $\mathrm{Inv}(i) \subseteq D$. Thus if $i \in R_\delta \cap D^c$ then $i \in S_\delta$. Hence

$$|R_\delta \cap D^c| \ \leq \ |S_\delta| \ \leq \ \mathrm{Ed}(\sigma)\left(\frac{1}{\delta} - 1\right)$$

□

Both bounds are asymptotically tight. For the first bound, let $k < n/4$ and take the permutation

$$\pi = k+1, \cdots, n/2, n, \cdots, n-k+1, 1, \cdots, k, n/2+1, \cdots, n-k$$

Here $\mathrm{Ed}(\pi) = 2k$ whereas $|R| = k$. For the second bound, let $k < \delta n$. Consider the permutation

$$\sigma = n, n-1, \cdots, n-k+1, 1, 2 \cdots, n-k$$

One can verify in this case that $\mathrm{Ed}(\sigma) = k$, whereas $|R_\delta| = k/\delta - 2$.

5.2 The Algorithm

This suggests a naive algorithm for estimating $\mathrm{Ed}(\sigma)$: for each interval $I = [j, i-1]$ ending in i, we sample $O(\log i)$ elements from I and test whether a majority of them lie in $\mathrm{Inv}(i)$. To implement this in a streaming algorithm, we maintain a bucket of elements sampled on the fly. This allows us to generate independent samples from each interval, samples from different intervals however will not be independent.

The algorithm reads the inputs from left to right, one element at a time. It will be convenient to say that the algorithm reads element $\sigma(i)$ at time i. The algorithm maintains a sample of the elements already seen in a bucket B. The fact that $\sigma(i)$ is retained in the bucket is denoted by $i \in B$, but note that the algorithm actually maintains in B a record of the tuple $\langle < i, \sigma(i) \rangle$. The algorithm keeps updating the bucket of samples so as to maintain the following distribution at each time i:

$$\Pr[j \in B \text{ at time } i\ ] = \min\left(1, \frac{C\log(2i)}{|j-i|}\right), \quad \text{for all } j < i.$$

We denote this probability by $p(j,i)$, and define it to be 1 for $j = i$. Note that for any j, $p(j,i) \geq p(j, i+1)$. Assume that we have the right distribution at time i. To achieve it at time $i+1$, we add the element $\sigma(i)$ to the bucket, and for each element $\sigma(j)$ already in the bucket $(j < i)$, we retain it with probability $p(j,i)/p(j,i+1)$.

The following proposition upper bounds the size of the sample that is retained, which immediately implies a similar bound on the space (storage requirement) of the algorithm.

Proposition 16 *At time i, $E[|B|] \leq C\log^2(2i)$. Further $|B| = O(\log^2 i)$ with probability $i^{-C'}$.*

Proof: Let X_j be the indicator variable for index j being in the bucket at time i. Note that the various X_is are independent and $|B| = \sum_{j<i} X_j$ Since

$$\Pr[X_j] \;=\; \min\left(1, \frac{C\log i}{|i-j|}\right)$$

$$E[|B|] \;=\; \sum_{j<i} \min\left(1, \frac{C\log(2i)}{|i-j|}\right) \;=\leq \sum_{j<i} \frac{C\log(2i)}{|i-j|} \;\leq\; C\log^2(2i)$$

The high concentration claim can be proved by a martingale argument. □

We next describe a procedure using B to test whether a near-majority of elements from $I_j = [j, i-1]$ lie in $\mathrm{Inv}(i)$.

```
TestBucket$(j,i)$

  Set $S_j \leftarrow \emptyset$.
  For $k \in [j, i-1]$,
     If $k \in B$ add it to $S_j$ with probability $p(j,i)/p(k,i)$.
  If at least $(\frac{1}{2} - \epsilon)$ fraction of $S_j$ lies in $\mathrm{Inv}(i)$, return Fail
  Else return Pass
```

The set S_j is our set of samples from I_j. It is easy to see that for all $k \in I_j$, we have $\Pr[k \in S_j] = p(j,i)$; furthermore, the events for different k (but the same j and i) are independent. We use this to show that the ratio $\frac{|S_j \cap \mathrm{Inv}(i)|}{|S_j|}$ is a fairly good approximation to $\frac{|I_j \cap \mathrm{Inv}(i)|}{|I_j|}$. We bound the error probability of the test by $(2i)^{-O(C)}$ where the constant in the $O(C)$ depends on ϵ. However this can be compensated for by choosing C appropriately.

Lemma 17 *If a majority of I_j lies in $\mathrm{Inv}(i)$, i.e. $\frac{|I_j \cap \mathrm{Inv}(i)|}{|I_j|} > 1/2$, then the probability TestBucket(j, i) returns Fail is at least $1 - (2i)^{-O(C)}$.*

Proof: Suppose $|I_j \cap \mathrm{Inv}(i)| > \frac{|I_j|}{2}$. Hence

$$E[|S_j|] = \sum_{k \in I_j} \frac{C \log(2i)}{|j-i|} = C \log(2i)$$

$$E[|S_j \cap \mathrm{Inv}(i)] = \sum_{k \in I_j \cap \mathrm{Inv}(i)} \frac{C \log(2i)}{|j-i|} \geq \frac{1}{2} C \log(2i)$$

One can show using Chernoff bounds that with probability $1 - i^{-O(C)}$,

$$|S_j| \leq (1 + \epsilon/2) C \log(2i)$$

$$|S_j \cap \mathrm{Inv}(i)| \geq (1/2 - \epsilon/2) C \log(2i)$$

In this case we have $|S_j \cap \mathrm{Inv}(i)| \geq (1/2 - \epsilon)|S_j|$ hence Test(B, i) will return Fail. $\Box$

Lemma 18 *If less than $(1/2 - 3\epsilon)$ fraction of I_j lies in $\mathrm{Inv}(i)$, i.e. $\frac{|I_j \cap \mathrm{Inv}(i)|}{|I_j|} < (1/2 - 3\epsilon)$, then the probability TestBucket(j, i) returns Pass is at least $1 - (2i)^{-O(C)}$.*

Proof: Suppose $|I_j \cap \mathrm{Inv}(i)| < (1/2 - 3\epsilon)|I_j|$. Hence

$$E[|S_j|] = \sum_{k \in I_j} \frac{C \log(2i)}{|j-i|} = C \log(2i)$$

$$E[|S_j \cap \mathrm{Inv}(i)] = \sum_{k \in I_j \cap \mathrm{Inv}(i)} \frac{C \log(2i)}{|j-i|} \leq (1/2 - 3\epsilon) C \log(2i)$$

One can show using Chernoff bounds that with probability $1 - (2i)^{-O(C)}$ the following bounds hold,

$$|S_j| \geq (1 - \epsilon) C \log(2i)$$

$$|S_j \cap \mathrm{Inv}(i)| \leq (1/2 - 2\epsilon) C \log(2i)$$

Now we have $|S_j \cap \mathrm{Inv}(i)| \leq (1/2 - \epsilon)|S_j|$ hence Test(B, i) will return Pass. $\Box$

We now describe the algorithm to estimate the Ulam distance using Test B(i, j). We maintain an estimate d which is initialized to $d = 0$. For each element i, we run Test B(i, j) for $j < i$. If one of them returns fail, we increment d. We update the bucket B and move to input $i + 1$. Let $\hat{R}$ denote the set of indices i which cause d to increase.

Lemma 19 *For every i, with probability $1 - \sum_i (2i)^{-O(C)}$ the following inclusion holds*

$$R \subseteq \hat{R} \subseteq R_{1/2 - 3\epsilon}$$

Proof: Assume that $i \in R$ and let j be a witness to this. Then by Lemma 17 running Test B(i,j) will return Fail with probability $1-(2i)^{-O(C)}$. Hence $\Pr[i \notin \hat{R}] \le (2i)^{-O(C)}$.

Assume on the other hand that $i \notin R_{1/2-3\epsilon}$. Then for every $j < i$, fewer than $1/2 - 3\epsilon$ elements in $[j, i-1]$ belong to $\mathrm{Inv}(i)$. By applying Lemma 18 and taking union bound over all i such intervals, the chance that Test B(i,j) returns Fail on any of these intervals is $(2i)^{-O(C)}$. Hence $\Pr[i \in \hat{R}] \le (2i)^{-O(C)}$.

□

Hence the inclusions hold with probability $1 - \sum_i (2i)^{-O(C)}$, which can be made larger than $1-\delta$ for any fixed $\delta > 0$ that we desire, by taking C to be a sufficiently large constant. By Lemma 12, we have $|R| \ge \mathrm{Ed}(\sigma)/2$ and $|R_{1/2-3\epsilon}| \le \frac{2}{1-6\epsilon}\,\mathrm{Ed}(\sigma)$. Hence with probability $1-\delta$ we get a $4+\epsilon'$ approximation to $\mathrm{Ed}(\sigma)$.

The description above performs Test B(i,j) for every $j < i$, so the update time for step i is linear in i. We can reduce the update time to $O(\log^3 i)$ by performing the test only for those $j \in B$. This can be further reduced to $O(\log^2 i)$ using an idea from [ACCL04]. We only try js for which the length of the interval $[j, i-1]$ changes in scales of $1+\epsilon_1$. More precisely, take $T'(i) = \{1, (1+\epsilon_1), \cdots, i\}$ and let $T(i) = \{j < i \ s.t.\ j - i \in T'(i)\}$.

Proposition 20 *If $i \in R$, then there exists $j' \in T(i)$ such that at least $(1/2 - \epsilon_1)$ fraction of elements from $[j', i-1]$ lie in $Inv(i)$.*

Proof: Let j be a witness to the membership of i in R. Hence a majority of elements from $[j, i-1]$ are in $Inv(i)$. Pick the smallest $j' \in T(i)$ such that $j' < j$. It follows that $|j - i| \le |j' - i| \le (1+\epsilon_1)|j-i|$. Hence at least $\frac{1}{2(1+\epsilon_1)} > (1/2 - \epsilon_1/2)$ fraction of elements from $[j', i]$ belong to $Inv(i)$.
□

We can choose ϵ_1 so that the analysis of Lemma 17 goes through even with the weaker assumption that $I_j \cap \mathrm{Inv}(i) \ge (1/2 - \epsilon_1/2)|I_j|$. We summarize the entire algorithm below.

Algorithm UlamDist(σ)

```
Set $d = 0$, bucket $B$ is empty.
For $i \in [1, n]$,
    Update bucket B.
    For each $j \in T(i)$
        If Test B$(j, i)$ returns Fail,
            Set $d = d + 1$.
            Skip to $i + 1$.
Output d.
```

Theorem 21 *Algorithm* **UlamDist**(σ) *computes a $4+\epsilon$ approximation to* $\mathrm{Ed}(\sigma)$ *with probability $1-\delta$ for arbitrarily small constant ϵ and δ. The space used is $O(\log^2 n)$ where n is the length of the data stream. The update time for element i is $O(\log^2 i)$.*

Note that if the length of the data stream n is known to us in advance, then we set the distribution of samples in the bucket to be

$$\Pr[j \in B] = p(j,i) = \min\left(1, \frac{C\log(2n)}{|j-i|}\right)$$

In this case, all the bounds stated above hold with probability $1 - n^{-O(1)}$. the space used and the update time are $O(\log^2 n)$.

5.3 Detailed description

Let describe the algorithm in more detail. $C > 0$ is a parameter to be determined. The mathematical analysis requires that it is larger than a suitable constant. Recall that we defined $p(j,i) = \min(1, \frac{C_2 \log n}{i-j})$. $\epsilon_1 > 0$ is a parameter that depends on the desired accuracy ϵ (e.g., $\epsilon_1 = \epsilon/3$ suffices). To simplify notation, we will write explicitly rounding operations, but throughout, all non-integral numerical values should be rounded downwards.

Algorithm UlamDist(σ)

```
1.  Initialize the bucket $B$ to be empty and set $d \leftarrow 0$
2.  For each $i = 1, 2, \ldots, n$
3.      Read the $i$th element (i.e.  $\sigma(i)$) from the input
4.      Remove from $B$ each tuple $\langle j, \sigma(j) \rangle$ independently with probability $1 - \frac{p(j,i)}{p(j,i-1)}$
5.      Add to $B$ the tuple $\langle i, \sigma(i) \rangle$
6.      For each $j' = 0, 1, \ldots, \log_{1+\epsilon_1} i$
7.         If TestBucket$(i - (1+\epsilon_1)^{j'}, i)$ returns Fail
8.             Set $d \leftarrow d + 1$ and continue to $i + 1$ at step 2
9.  Output $d$.
```

Procedure TestBucket(j, i)

```
1.  Initialize the set $S$ to be empty
2.  For every tuple $\langle k, \sigma(k) \rangle$ in $B$
3.    If $j \le k \le i - 1$
4.      then add $\langle k, \sigma(k) \rangle$ to $S$ with probability $p(j,i)/p(k,i)$.
5.  If at least $(\frac{1}{2} - \epsilon_1)$ fraction of the tuples $\langle k, \sigma(k) \rangle$ in $S$ satisfy $\sigma(k) > \sigma(i)$
6.    then return Fail
7.  return Pass
```

Efficient sampling. We suggest a simple and efficient method for the random sampling of elements (both to maintain the bucket B in Algorithm UlamDist and to select the sample S in procedure TestBucket). When element j is first seen, the algorithm chooses for it a threshold $Z_j \in [0,1]$ uniformly at random.[1] Now whenever we wish to sample the element $\sigma(j)$ with probability p, we do it by testing whether $Z_j \le p$. In particular, the element $\sigma(j)$ is retained in the bucket B at all times i for which $Z_j \le p(j,i)$, so line 4 in UlamDist in changed to removing element j if $Z_j > p(j,i)$; thus, the random value Z_j provides an "expiration time" at which element j is to be discarded from B. Similarly, line 4 of procedure TestBucket is changed so that element k is added to S if $Z_k \le p(j,i)$. Clearly, the algorithm needs to retain the threshold Z_j only for elements $\sigma(j)$ that are retained in the bucket B, and thus these thresholds increase the storage requirement only by a small constant factor. The important aspect is that the different Z_j are independent; it

[1]In practice, Z_j will be determined up to some precision, which may be increased during execution.

does not matter that the same Z_j is used for different samples S in procedure TestBucket, since we apply a union bound over the different executions of this procedure.

The advantage of this threshold value is that there is that it only one call to a random (or pseudo-random) number generator every time a new element is input. In addition, it avoids probability of the sort $p(j,i)/p(j,i-1)$ which are very close to 1 and thus it is effectively more expensive to draw events according to such probabilities.

5.4 Alternative implementation

We describe an alternative algorithm in which we maintain $O(\log n)$ smaller buckets. In the sequel, $C_1, C_2, C3 > 0$ are three parameters to be determined. The mathematical analysis requires that they are larger than a suitable constant. We also define $q(j,i) = \min(1, \frac{C_2}{i-j})$. $\epsilon_1 > 0$ is a parameter that depends on the desired accuracy ϵ (e.g., $\epsilon_1 = \epsilon/3$ suffices). To simplify notation, we will write explicitly rounding operations, but throughout, all non-integral numerical values should be rounded downwards.

Algorithm UlamDistAlt(σ)

```
1.  Set $d \leftarrow 0$ and $t \leftarrow C_1 C_3 \log n$ and initialize $t$ buckets $B_1, \ldots, B_t$ to be empty
2.  For each $i = 1, 2, \ldots, n$
3.    Read the $i$th element (i.e. $\sigma(i)$) from the input
4.    For each $s = 1, \ldots t$
5.      Remove from $B_s$ each tuple $\langle j, \sigma(j) \rangle$ independently with probability $1 - \frac{q(j,i)}{q(j,i-1)}$
6.      Add to $B_s$ the tuple $\langle i, \sigma(i) \rangle$
7.    For each $j' = 0, 1, \ldots, \log_{1+\epsilon_1} i$
8.      Set $S$ to be the empty set
9.      For each $s = 1, \ldots t$
10.        Add to $S$ the element returned by SampleOneBucket($B_s, j', i$)
11.      If at least $(\frac{1}{2} - \epsilon_1)$ fraction of the tuples $\langle k, \sigma(k) \rangle$ in $S$ satisfy $\sigma(k) > \sigma(i)$
12.      then $d \leftarrow d + 1$ and continue to $i + 1$ at step 2
13.  Output $d$.
```

Procedure SampleOneBucket(B_s, j, i)

```
1.  Initialize the set $S'$ to be empty
2.  For every tuple $\langle k, \sigma(k) \rangle$ in $B$
3.    If $j \le k \le i - 1$
4.      then add $\langle k, \sigma(k) \rangle$ to $S'$ with probability $q(j,i)/q(k,i)$.
5.  return a random element from $S$ (if any)
```

As before, a simple and efficient sampling is possible by retaining for each element in each bucket a random threshold value Z_{sj}. We stress that it is important choose independently the thresholds for different buckets B_s.

6 A Lower Bound for Approximating the LIS

We are interested in the one-way communication complexity of functions f in the following model:

- There are k players $P_1, \cdots, P_k$, each player is given part of the input.
- Player P_i sends a message to P_{i+1}, in the order $P_1, \cdots, P_{k-1}$. Players might choose to not send anything when their turn comes.
- The protocol terminates when some player knows the value of f.

We are interested in the maximum communication complexity of a function denoted by $M_k(f)$ and the total communication complexity denoted by $T_k(f)$.

Given a sequence of numbers $x = x_1, \cdots, x_k$ from the alphabet $[m]$, define the (promise) function

$$\begin{aligned} h(x) &= 0 \quad if \quad LIS(x) = 1 \\ h(x) &= 1 \quad if \quad LIS(x) \geq k/2 \end{aligned}$$

We are interested in the communication complexity of h when player P_i gets number x_i as input.

The motivation for this comes from the following function g defined on k^2 numbers:

$$g(x_1^1, \cdots, x_k^1, \cdots, x_1^k, \cdots, x_k^k) = \vee_{i=1}^k h_i(x_1^i, \cdots, x_k^i)$$

In other words, g is the OR of k disjoint copies of h. Consider a protocol for computing g where player j receives the numbers x_j^i for $i = 1, \cdots k$, i.e. he receives the j^{th} number for each copy of h.

Lemma 22 *$M_k(g)$ is a lower bound on the space used by any deterministic one-pass streaming algorithm that approximates the LIS.*

Proof: Given number x_j^i, define a number $y_j^i = m(i-1) + x_j^i$. Thus the numbers $y_1^i, \cdots, y_k^i$ lie in the interval $[m(i-1)+1, \cdots, mi]$. From the sequence $y = y_1^1, \cdots, y_1^k, \cdots, y_k^1, \cdots, y_k^k$. It is easy to verify the following condition:

$$\begin{aligned} g(x) = 0 &\Rightarrow LIS(y) = k \\ g(x) = 1 &\Rightarrow LIS(y) \geq 1.5k \end{aligned}$$

Thus a streaming algorithm using space S would imply a communication protocol to compute $g(x)$ using maximum communication S. $\square$

The following claim would imply an $\Omega(\sqrt{n})$ lower-bound for LIS:

Claim 23 *$M_k(g) = \Omega(k)$.*

An approach to proving this claim might be to show:

Claim 24
- *$T_k(h) = \Omega(k)$.*
- *$T_k(g) = \Omega(k^2)$.*

Indeed if $T_k(g) = \Omega(k^2)$, then $M_k(g) = \Omega(k)$. Since g is the OR of k disjoint copies of h, it is reasonable to hope that $T_k(g) = kT_k(h)$. This is not implied by any direct-sum theorem that we know of. However, one could hope that if we do prove an $\Omega(k)$ bound for $T_k(h)$, the same proof would give a bound for $T_k(g)$. We will present a proof along these lines, which however requires $m = 2^k$. Finally, it appears that h is a simpler function than g, so *without a non-trivial lower-bound for h, such a bound for g seems hopeless.*

6.1 A Lower-bound on $T_k(h)$

We show an $\Omega(k)$ lower bound for how many players have to send messages in the worst case. We will ignore the issue of message size, players can send messages of any size. There is a protocol for this problem where only the first $k/2+1$ players send messages. We will show that this is in fact tight.

Theorem 25 *In any protocol to compute h, there are inputs x where at least $k/2+1$ players send messages.*

Proof: The proof uses integers lying in the range $\{0, \cdots, 3^k - 1\}$. Using induction on j, we construct inputs $x = x_1 \cdots x_j$ and $y = y_1 \cdots y_j$ reaching player P_{j+1} with the following properties:

1. x and y have the same transcript.
2. $LIS(x) = 1$; i.e. x is a decreasing sequence ending at some number a.
3. If t players have not sent messages, then y contains an IS of length t terminating at b where $b > a$. Note that this need not be the LIS of y.

At each stage, the player will be given inputs from an interval $[a, b]$. Specifically, the player will get either $c_0 = \frac{2a_i+b_i}{3}$, or $c_1 = \frac{a_i+2b_i}{3}$ as the input. To begin with, let $a = 0, b = 3^k$.

- **Base Case:** We give P_1 two inputs, c_0 and c_1. If for one of these two inputs (call it c), P_1 sends a message to P_2, we set $x_1 = y_1 = c$. Now we set $a = c$, b stays unchanged. Conditions 1-3 are satisfied by this choice of a, b where $t = 0$.

  Else, P_1 is silent on both inputs. Now set $x = c_0$ and $y = c_1$. Also let $a = c_0$ and $b = c_1$. Conditions 1-3 are satisfied by this choice of a, b where $t = 1$.

- **Inductive case:** Assume that t players have been silent so far. Then conditions 1-3 are satisfied for some inputs x and y. Again there are two cases to consider. If P_j sends a message on some input c, then we set $a = c$, and b stays the same. We set $x \leftarrow x \cdot c$ and $y \leftarrow y \cdot c$. Note that y now contains a longer increasing sequence terminating at c, but we are only concerned with the increasing sequence that terminates at b. Conditions 1-3 are satisfied for t.

  If player P_j does not send a message on both inputs, then we take $a = c_0, b = c_1$. We set $x \leftarrow x \cdot c_0$ and $y \leftarrow y \cdot c_1$. Now the conditions 1-3 are satisfied for $t+1$.

Now clearly if $t \geq k/2$ then the protocol is wrong, so the Lemma follows. □

A corollary of this is that $T_k(g) = \Omega(k)$.

6.2 A Lower-bound on $M_k(g)$

We show that one can extend this argument to get a lower-bound on $M_k(g)$.

Theorem 26 $M_k(g) = \Omega(k)$.

Proof: Assume that we have a protocol for g with max. communication bounded by δk for some constant $\delta > 0$ to be chosen later. Player j receives the j^{th} argument for each problem h_i, in other words he gets $x_j^1, \cdots, x_j^k$. Using induction on j, we construct inputs x and y reaching player P_{j+1} with the following properties:

1. x and y have the same transcript.
2. $LIS(x^i) = 1$ for $x^i = x_1^i, \cdots, x_j^i$; i.e. x^i is a decreasing sequence ending at some number a_i.
3. The sequence $y^i = y_1^i, \cdots, y_j^i$ contains an IS of length t_i terminating at b_i where $b_i > a_1$.
4. $\sum_i t_i \geq \epsilon jk$ where ϵ is some explicit function of δ.

We repeat the same inductive construction of inputs for k copies of the function h. For each co-ordinate i, the input given to player j comes from the interval $[a_i, b_i]$, it takes one of two possible values, $\{\frac{2a_i+b_i}{3}, \frac{a_i+2b_i}{3}\}$. There are a total of 2^k possible inputs to player P_j, which we identify with the bit strings in $\{0,1\}^k$. Since the max. communication is at most δk, by the pigeonhole principle, he sends the same message on $2^{(1-\delta)k}$ inputs. It particular, there are two strings r, s reaching the same state where $r_i = 0$ and $s_i = 1$ for ϵk co-ordinates and where ϵ is some explicit function of δ. Call such co-ordinates 'bad'. We update x by extending it by the input corresponding to r, and y by the input corresponding to s.

For every bad co-ordinate i, we can take $x_j^i = \frac{2a_\ell+b_\ell}{3}$, whereas $y_j^i = \frac{a_\ell+2b_\ell}{3}$, and set $a_i = \frac{2a_i+b_i}{3}$, $b_i = \frac{a_i+2b_i}{3}$. In other words, for bad co-ordinates we can increase the length of the LIS in y^i. For the other co-ordinates, we update the value of a_i depending on whether r_i is 0 or 1, but leave b_i unchanged. For such co-ordinates, even though there is a longer increasing sequence in y^i, we ignore it.

Since some ϵk increasing sequences are extended at every step, at the end we get $\sum_i t_i \geq \epsilon k^2$ for some explicit constant $\epsilon > 0$. In particular, for some i, we get $t_i > \epsilon k$. Repeating the reduction of Lemma 22, we get a δk lower bound on the space required for a $(1+\epsilon)$ approximation to the LIS. (The constant ϵ may not be 1/2, but it is some explicit constant that only depends on δ.)

□

6.3 Reducing the Size of the Alphabet

The above proof is modeled after Ajtai's lower-bound for approximate counting. We reduced each state to a single interval $[a, b]$ for the function h and a rectangle $[a_1, b_1] \times \cdots \times [a_k, b_k]$ for the function g. However, it appears that this simplification causes us to require an exponential alphabet of size $m = 3^k$, since roughly the interval size can reduce to $1/3^{rd}$ at every step.

To get around this seems to require a more detailed description of each state. The decreasing sequences reaching a state can still be represented by a single number, that represents the end point of the 'best' decreasing sequence. But for the increasing sequences reaching a state, we need to keep track of the entire profile of the sequences i.e. for every length ℓ, we should keep track of the best increasing sequence of length ℓ reaching that state. One might hope to show similar bounds to the above, even when $m = k^{O(1)}$.

References

[ACCL04] Nir Ailon, Bernard Chazelle, Seshadhri Comandur, and Ding Liu. Estimating the distance to a monotone function. In 8^{th} *International Workshop on Randomization and Computation, (RANDOM'04)*, pages 229–236, 2004.

[AD99] David Aldous and Persi Diaconis. Longest Increasing Subsequences: From Patience Sorting to the Baik-Deift-Johansson theorem. *BAMS: Bulletin of the American Mathematical Society*, 36, 1999.

[AJKS02] Mikls Ajtai, T.S. Jayram, Ravi Kumar, and D. Sivakumar. Approximate counting of inversions in a data stream. In *Proceedings of the* 34th *Annual ACM Symposium on Theory of Computing (STOC'02)*, pages 370–379, 2002.

[Ajt02] Miklos Ajtai. Unpublished manuscript, 2002.

[BYJKS02] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. 2002.

[CMS01] G. Cormode, S. Muthukrishnan, and S. C. Sahinalp. Permutation editing and matching via embeddings. In *Proceedings of* 28th *International Colloquium on Automata, Languages and Programming (ICALP'01)*, pages 481–492, 2001.

[EKK$^+$00] Funda Ergun, Sampath Kannan, Ravi Kumar, Ronitt Rubinfeld, and Mahesh Viswanathan. Spot-checkers. *Journal of Computing and System Sciences*, 60(3):717–751, 2000.

[FLN$^+$02] Eldar Fischer, Eric Lehman, Ilan Newman, Sofya Raskhodnikova, Ronitt Rubinfeld, and Alex Samorodnitsky. Monotonicity testing over general poset domains. In *Proceedings of the* 34th *Annual ACM Symposium on Theory of Computing (STOC'02)*, pages 474–483, 2002.

[GGL$^+$00] Oded Goldreich, Shafi Goldwasser, Eric Lehman, Dana Ron, and Alex Samorodnitsky. Testing monotonicity. *Combinatorica*, 20(3):301–337, 2000.

[GZ03] Anupam Gupta and Francis Zane. Counting inversions in lists. In *Proceedings of the* 14th *ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*, pages 253–254, 2003.

[Vit85] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.