# Estimating the Longest Increasing Subsequence in Nearly Optimal Time

Alexandr Andoni
*Columbia University*
New York, USA
andoni@cs.columbia.edu

Negev Shekel Nosatzki
*Columbia University*
New York, USA
ns3049@columbia.edu

Sandip Sinha
*Columbia University*
New York, USA
sandip@cs.columbia.edu

Clifford Stein
*Columbia University*
New York, USA
cliff@cs.columbia.edu

***Abstract*—Longest Increasing Subsequence (LIS) is a fundamental statistic of a sequence, and has been studied for decades. While the LIS of a sequence of length $n$ can be computed exactly in time $O(n \log n)$, the complexity of estimating the (length of the) LIS in sublinear time, especially when LIS $\ll n$, is still open.**

**We show that for any $n \in \mathbb{N}$ and $\lambda = o(1)$, there exists a (randomized) *non-adaptive* algorithm that, given a sequence of length $n$ with LIS $\geq \lambda n$, approximates the LIS up to a factor of $1/\lambda^{o(1)}$ in $n^{o(1)}/\lambda$ time. Our algorithm improves upon prior work substantially in terms of both approximation and run-time: (i) we provide the first sub-polynomial approximation for LIS in sub-linear time; and (ii) our *run-time complexity* essentially matches the trivial *sample complexity* lower bound of $\Omega(1/\lambda)$, which is required to obtain any non-trivial approximation of the LIS.**

**As part of our solution, we develop two novel ideas which may be of independent interest. First, we define a new Genuine-LIS problem, in which each sequence element may be either genuine or corrupted. In this model, the user receives unrestricted access to the actual sequence, but does not know a priori which elements are genuine. The goal is to estimate the LIS using genuine elements only, with the minimal number of tests for genuineness. The second idea, *Precision Tree*, enables accurate estimations for composition of general functions from "coarse" (sub-)estimates. *Precision Tree* essentially generalizes classical precision sampling, which works only for summations. As a central tool, the *Precision Tree* is pre-processed on a set of samples, which thereafter is repeatedly used by multiple components of the algorithm, improving their amortized complexity.**

***Index Terms*—sublinear algorithms, approximation algorithms, randomized algorithms, longest increasing subsequence, string algorithms**

## I. Introduction

Longest Increasing Subsequence (LIS) is a fundamental measure of a sequence, and has been studied for decades. Near linear-time algorithms have been known for a long time, for example, the Patience Sorting algorithm [29], [32] finds a LIS of a sequence of length $n$ in time $O(n \log n)$. The celebrated Ulam's problem asks for the length of a LIS in a random permutation; see discussion and results in [2]. LIS is also an important special case of the problem of finding a Longest Common Subsequence (LCS) between two strings, as LIS is LCS when one of the strings is monotonically increasing. Recently, there has been significant progress in approximation algorithms for LCS of two or more strings [12], [15], [22], [42]. Moreover, LIS is often a subroutine in LCS algorithms: for example, when strings are only mildly repetitive [27, Chapter 12], or more recently in approximation algorithms [28], [38]. Longest increasing subsequences have multiple applications in areas such as random matrix theory, representation theory, and physics [2], and the related LCS problem also has multiple applications in bioinformatics, and is used for data comparisons such as in the `diff` command.

In the quest for faster algorithms, researchers started studying whether we can estimate the length of a LIS (denoted LIS as well) in *sublinear time*. An early version of this question underpins one of the first sublinear-time algorithms: to test whether an array is sorted, or monotonically increasing [18] — i.e., whether the length of the LIS is $n$ or is at most $(1-\epsilon)n$. [18] gave a $O(\log n/\epsilon)$ time algorithm, and this running time was later shown to be tight [1], [20]. Since then, there have been numerous influential results on testing monotonicity and other similar properties; see, e.g., [7], [9]–[11], [14], [16], [19], [40], [41], [45] and the book [25].

While monotonicity results focus on the case when LIS $\approx n$, the case when LIS $\ll n$ has seen much less progress. The first result for this regime [44] shows how to $(1+\epsilon)$-approximate the length when the LIS is still large: they distinguish the case when LIS $\geq \lambda n$ from the case when LIS $\leq (\lambda - \epsilon)n$ (for any $\lambda > 0$) using $(1/\epsilon)^{O(1/\epsilon)} \log^{O(1)} n$ time, which only gives a $(1+\epsilon)$-factor approximation in truly sublinear time if $\lambda = \Omega(\log\log n/\log n)$. For arbitrary $\lambda < 1$, assuming that LIS $\geq \lambda n$ [42] gave an algorithm that achieves $O(1/\lambda^3)$-factor approximation of

Alexandr Andoni: Research supported in part by NSF grants (CCF-1617955, CCF-1740833, CCF-2008733), and Simons Foundation (#491119).
Negev Shekel Nosatzki: Research supported in part by NSF grants (CCF-1617955 and CCF-1740833), and Simons Foundation (#491119).
Sandip Sinha: Research supported by NSF grants CCF-1714818, CCF-1822809, IIS-1838154, CCF-1617955, CCF-1740833, and by the Simons Collaboration on Algorithms and Geometry.
Clifford Stein: Research partly supported by NSF Grants CCF-1714818 and CCF-1822809.

LIS in $\tilde{O}(\sqrt{n}/\lambda^7)$ time.[1] Recently, [34] improved upon this result by presenting an algorithm with approximation $O(1/\lambda^\epsilon)$ and runtime $O(n^{1-\Omega(\epsilon)}(\log n/\lambda)^{O(1/\epsilon)})$. Independently, [37] obtained a non-adaptive $O(1/\lambda)$-approximation algorithm using $\tilde{O}(\sqrt{r}/\lambda^2)$ queries, where $r$ is the number of distinct values. The authors of [37] also proved a lower bound, showing that any *non-adaptive* algorithm that estimates the LIS to within an additive error of $\epsilon n$ requires $(\log n)^{\Omega(\log(1/\epsilon))}$ queries.

To put the above results into context, contrast LIS with the problem of estimating the weight of a binary vector of length $n$: when the weight is at least $\lambda n$, we can approximate the weight up to a factor of $1+\epsilon$ by sampling $O_\epsilon(1/\lambda)$ positions, which is optimal. So far, one cannot rule out that a similar performance is achievable for estimating LIS too, in fact for $\lambda$ as large as $1/\log n$. The aforementioned prior results not only have approximation factors that are polynomial in $1/\lambda$ but also time / sample complexities that are worse by a polynomial factor in $n$ or $1/\lambda$. Hence the following guiding question remains open:

*Can we estimate the length of* LIS *in essentially the time needed to estimate the weight of a binary vector?*

### A. Our contributions

In this paper we come close to answering the above question in the affirmative, by obtaining an algorithm that runs in time near-linear in $1/\lambda$, and achieves sub-polynomial in $1/\lambda$ approximation. We show the following:

**Theorem I.1** (Main theorem). *For $n \in \mathbb{N}$, and any $\lambda = o(1)$, there exists a (randomized) non-adaptive algorithm that, given a sequence of length $n$ with* LIS $\geq \lambda n$*, approximates the length of* LIS *up to a $1/\lambda^{o(1)}$ factor in $O\left(\frac{1}{\lambda} \cdot n^{o(1)}\right)$ time with high probability.*

We find this result quite surprising, as one may guess that when $\lambda \approx 1/\sqrt{n}$ (which is the LIS length of a random permutation), one would need to read essentially the entire sequence (implying a $\Omega(1/\lambda^2)$ lower bound on the number of queries needed). On the contrary, when $\lambda \approx 1/\sqrt{n}$, our run-time (and hence sample complexity) nearly matches the *streaming complexity* from [17], [26] (albeit with worse approximation).

**Technical Contributions.** Our main technical contributions are three-fold (see technical overview in Section III):

- We define (the LIS problem in) a new sublinear computational model, which we call Genuine-LIS and which may be of independent interest. In the Genuine-LIS problem, we are given a sequence $y \in \mathbb{N}^n$, with a caveat that only some of the elements of $y$ are "genuine" and the others are "corrupted", a property we can test an element for. The goal is to estimate the length of LIS *among the genuine* elements of $y$, using as few tests as possible, while the values of $y$ are known to the algorithm "for free".
- We show how one can efficiently reduce the LIS problem to Genuine-LIS and vice versa. These reductions together constitute the backbone of our recursive algorithm.
- To obtain the promised query complexity, we develop a new data structure, that we call a Precision-Tree. This data structure samples non-adaptively (but non-uniformly) elements of the input $y$ and preprocesses them for efficient operations on the sampled elements. This part is the only part of the algorithm that samples the input string — the aforementioned recursive calls of LIS and Genuine-LIS problems access this data structure only. Furthermore, the data structure allows one to compute accurate estimations for composition of general functions from "coarse" (sub-)estimates — a requirement for our recursive approach.

While this paper makes progress in understanding the complexity of estimating the LIS, the following important question remains open:

**Open Question** ($(1+\epsilon)$-approximation)**.** *Does there exist an algorithm that, given a sequence of length $n$ with* LIS $\geq \lambda n$*, estimates the* LIS *up to a $1+\epsilon$ factor in $O\left(\frac{1}{\lambda} \cdot n^{o(1)}\right)$ time with probability 2/3?*

We believe that finding an improved approximation algorithm for the Genuine-LIS problem above may lead to a $(1+\epsilon)$ approximation algorithm for LIS.

We give a technical overview of our algorithm in Section III, after setting up preliminaries in Section II. Section IV contains the proof of the main theorem, assuming results proved in the full version of our paper [8]. In Section V, we develop the Precision-Tree data structure that is used to improve the sample and time complexity of the main algorithm. The guarantees of the two primary subroutines, as well as extensions of these algorithms in certain ways critical to our final application, are deferred to the full version [8].

### B. Related work

Computing the length of LIS has also been studied in the streaming model, where settling its complexity is a major open problem [39]. In this setting, the main question is to determine the minimum space required to estimate the length of LIS by an algorithm reading the sequence left-to-right. [17], [26] gave deterministic one-pass algorithms to $(1+\epsilon)$-approximate the LIS using $O(\sqrt{n})$ space, and matching lower bounds against deterministic algorithms

[1]The formal definition of an approximation algorithm here is one that has to return $\widehat{\text{LIS}}$ such that $\widehat{\text{LIS}} \leq$ LIS, and the approximation factor is LIS$/\widehat{\text{LIS}}$. Equivalently, one can conceptualize an $\alpha$-approximation algorithm (for $\alpha \geq 1$) as one that can distinguish the case when LIS $\geq \lambda n$ from the case when LIS $< \lambda n/\alpha$.

were given by [17], [21]. These lower bounds are derived using deterministic communication complexity lower bounds and provably fail to extend to randomized algorithms [13]. However, no randomized algorithm requiring $o(\sqrt{n})$ space is known for this problem either. We note that our algorithm can be used in the streaming setting, yielding streaming complexity $O(n^{1/2+\epsilon})$ for approximation $n^{o(1)}$.

There has been much more success on the "complement" problem of estimating the distance to monotonicity, i.e., $d_m := n - \mathrm{LIS}$, in both the sublinear-time and the streaming settings. In the random-access setting, the problem was first studied in [1], and later in [44], who gave an algorithm that achieves $(1+\epsilon)$-approximation in time $\mathrm{poly}(1/d_m, \log n)$ for any constant $\epsilon > 0$. These algorithms have been also used, indirectly, for faster algorithms for estimating Ulam distance [3], [36] and smoothed edit distance [4].

In the streaming setting, several results were obtained which achieve $O(1)$-approximation of $d_m$ using $\mathrm{polylog}(n)$ space [17], [26]. This culminated in a randomized $(1+\epsilon)$-approximation algorithm using $\mathrm{polylog}(n)$ space by Saks and Seshadhri [43], and a deterministic $(1+\epsilon)$-approximation algorithm using $\mathrm{polylog}(n)$ space by Naumovitz and Saks [35]. [35] also showed space lower bounds against $(1+\epsilon)$-approximation streaming algorithms of $\Omega(\log^2 n/\epsilon)$ (deterministic) and $\tilde{\Omega}(\log^2 n/\epsilon)$ (randomized).

The LIS problem has also been studied recently in other settings, such as the MPC and the fully dynamic settings. [30] gave a $(1+\epsilon)$-approximation, $O(1/\epsilon^2)$-round MPC algorithm for LIS whenever the space per machine is $n^{3/4+\Omega(1)}$. In the dynamic setting, a sequence of works [23], [33] culminating in [31], gave the first exact dynamic LIS algorithm with sublinear update time, and also gave a deterministic algorithm with update time $n^{o(1)}$ and approximation factor $1 - o(1)$. [24] showed conditional lower bounds on update time for certain variants of the dynamic LIS problem.

**Subsequent Work.** The main result from this paper was very recently used for estimating the Longest Common Subsequence (LCS) problem. In particular, [38] gave the first sub-polynomial approximation for LCS in linear time, invoking our algorithm as a sub-routine.

## II. Preliminaries

**Sequences and intervals.** Given a set $\Omega$ and $n \in \mathbb{N}$, a sequence $y = (y_1, y_2, \cdots, y_n) \in \Omega^n$ is an ordered collection of elements in $\Omega$. A block sequence $y \in \Omega^{n \times k}$ is a partially ordered collection of elements in $\Omega$. Abusing notation, we will also allow for some elements in a block to be "null" — for example, we write $y \in \mathbb{N}^{n\times k}$ to also mean $y \in \{\mathbb{N}, \perp\}^{n \times k}$, where $\perp$ is "null".

For $\ell \in [n]$, we say that $z = (z_1, \cdots, z_\ell)$ is a subsequence of $y$ of length $\ell$, and denote it by $\{y_{i_j}\}_j$, if there exist integers $1 \le i_1 < \cdots < i_\ell \le n$ such that $z_j = y_{i_j}$ for all $j \in [\ell]$. We refer to the indices $i_j$ as coordinates and the values $y_{i_j}$ as element values. For a block sequence $y$, we say that $z = (z_1, \cdots, z_\ell)$ is a subsequence of $y$ of length $\ell$, and denote it by $\{y_{w_j}\}_j$, if there exist integers $1 \le i_1 < i_2 < \cdots < i_\ell \le n$ such that $z_j \in y_{i_j,*}$ for all $j \in [\ell]$.

Define the interval space $\mathcal{I} \triangleq \{[a,b] \mid a, b \in \mathbb{N}, a \le b\} \cup \{[a,b) \mid a \in \mathbb{N}, b \in [1,\infty], a < b\}$. For $I \in \mathcal{I}$, we use $|I|$ to denote $|I \cap \mathbb{N}|$, i.e., the number of natural numbers contained in interval $I$. We often refer to an $x$-interval $X \in \mathcal{I} \cap 2^{[n]}$ as an interval of coordinates, and a $y$-interval $Y \in \mathcal{I}$ as an interval of element values.

For a block sequence $y \in \mathbb{N}^{n\times k}$ and a $y$-interval $Y \in \mathcal{I}$, we write $y \cap Y$ to denote the *multi-set* of elements in $y$ that are also in $Y$. Also, for $X \subseteq [n]$, $y(X)$ is the sequence with first coordinates restricted to $X$. We also define $y(X,Y) := y(X) \cap Y$.

**Monotonicity.** We define monotone sets as follows:

**Definition II.1** (Monotone sets)**.** *Fix a (potentially partial) ordered set* $(\Omega, <)$*. We say that a set* $P \subseteq \mathbb{N} \times \Omega$ *is* monotone *if for all* $(i,u), (j,v) \in P \times P$*, we have (i)* $i = j \Leftrightarrow u = v$*; and (ii)* $i < j \Leftrightarrow u < v$*.*

Note that this definition captures the notion of an increasing subsequence. In particular, for the standard notion of an increasing subsequence over natural numbers, we take $\Omega = \mathbb{N}$ and $<$ as the usual "less than" relation over $\mathbb{N}$ (a total order). However, we will need this more general definition to consider increasing subsequences over other partially ordered sets, like the space of intervals $\mathcal{I}$.

For a finite set $P \subset \mathbb{N} \times \Omega$, a longest increasing subsequence (LIS) of $P$ is a monotone set $Q \subset P$ of maximum cardinality. We often use $\mathsf{OPT}$ to refer to a particular LIS, and use $|\mathsf{OPT}|$ to denote its length.

**Distributions.** For $p \in [0,1]$, we use $\mathtt{Ber}(p)$ to denote the Bernoulli distribution with success probability $p$, and $\mathtt{Bin}(n,p)$ to denote the binomial distribution with parameters $n$ and $p$. By convention, we project $p$ to the range $[0,1]$ whenever $p > 1$ or $p < 0$.

We use "i.i.d. random variables" to mean that a collection of random variables is independent and identically distributed, and use "sub-sampling i.i.d. with probability $p$" to mean that each element is sampled independently with equal probability $p$.

**Operations on Vectors, Sets, Functions.** For a set $A \subset \mathbb{R}$ and a number $\alpha$, we define $A + \alpha := \{a + \alpha : a \in A\}$ and $\alpha A := \{\alpha\, a : a \in A\}$. We write $\log$ to denote binary logarithm.

The notation $\circ$ is used for function composition (i.e., $g \circ f(x) = g(f(x))$, and $\oplus$ is used for direct sum. We use the notation $*$ as argument of a function, by which we mean a vector of all possible entries. For example, $f(*)$ is a vector of $f(i)$ for $i$ ranging over the domain of $f$ (usually clear from the context).

We use $E_b(k)$ for $b, k \in \mathbb{N} \cup \{\infty\}$ to denote the set of powers of $b$ bounded by $k$, i.e., $E_b(k) \triangleq \{b^i \mid i \in \mathbb{N}\} \cap [1, k]$. We also define $E_b \triangleq E_b(\infty)$. We use $\mathbb{R}_+$ to denote the set of non-negative real numbers.

**Approximations.** Our constructions generate approximations which contain both multiplicative and additive terms. We use the following definition:

**Definition II.2** ($(\alpha, \beta)$-Approximation)**.** *For $\alpha \geq 1$ and $\beta > 0$, an $(\alpha, \beta)$-approximation $\hat{q}$ of a quantity $q$ is an $\alpha$-multiplicative and $\beta$ additive estimation of $q$, i.e., $\hat{q} \in [q/\alpha - \beta, q]$.*

The following fact shows that to obtain a multiplicative approximation which is a function of the LIS, it suffices to show $(\alpha, \beta)$-approximation.

**Fact II.3.** *Suppose we have an algorithm $\mathcal{A}$ that for any $n \in \mathbb{N}, \lambda \in (0, 1)$, for some $q \in (0, 1)$, outputs a $(1/\lambda^q, \lambda n)$-approximation for some unknown quantity $\ell \in [0, n]$ in time $t$. Then there exists an algorithm that outputs $\widehat{\ell} \in [\Omega(\delta^p), 1] \cdot \ell$ in time $t + O(1)$, where $p \triangleq \frac{q}{1-q}$, as long as $\ell \geq 2\delta n$.*

*Proof.* The algorithm calls $\mathcal{A}$ with parameter $\lambda = \delta^{1+p}$, and returns the same output. Then, the upper bound is immediate by definition of $(\alpha, \beta)$-approximation. For the lower bound, as long as $\ell \geq 2\delta n$, we have that:

$$\widehat{\ell} \geq \delta^{q(1+p)} \cdot \ell - \delta^{1+p} \cdot n \geq \delta^p \cdot \ell - \tfrac{1}{2}\delta^p \cdot \ell \geq \tfrac{1}{2}\delta^p \ell.$$

□

**Other notation.** Notation $\tilde{O}(\cdot)$ hides $\mathrm{polylog}(n)$ factors, while $O^*(\cdot)$ hides a factor of $n^{o(1)}$.

## III. Technical overview

Our starting point is the algorithm of [42] which achieves $O(1/\lambda^3)$ approximation in time $\tilde{O}(\sqrt{n}/\lambda^7)$. Let $\mathsf{OPT}$ be (the coordinates of) an optimum solution (LIS) with length $|\mathsf{OPT}| \geq \lambda n$. Their algorithm consists of 2 main steps:

1) Partition $[n]$ into $\sqrt{n}$ disjoint contiguous x-intervals $X_1, X_2, \ldots, X_{\sqrt{n}}$ of length $\sqrt{n}$ each. For $i \in [\sqrt{n}]$, let $y(X_i)$ be the restriction of the input sequence $y$ to $x$-interval $X_i$. The goal is to approximate the $y$-interval $Y_i \triangleq [s_i, \ell_i]$, where $s_i$ and $\ell_i$ are, respectively, the minimum and maximum values in $y(X_i \cap \mathsf{OPT})$. To do this, one samples $O(1/\lambda)$ elements in $X_i$, and generates $O(1/\lambda^2)$ *candidate $y$-intervals* (using pairs of sampled element values).
2) Generate a set of mutually disjoint *pseudo-solutions*, each of which is a sequence of $\Omega(\lambda\sqrt{n})$ candidate $y$-intervals that are monotone, i.e., all values in a candidate interval in x-interval $X_i$ are less than all values in a candidate interval in x-interval $X_j$ for all $i < j$. Estimate the quality of each pseudo-solution (the sum of LIS of the candidate intervals in it) by simply sampling $O(\log^{O(1)} n/\lambda^4)$ x-intervals and computing the LIS within relevant candidate intervals. Output the largest quality.

Their analysis proceeds as follows. For each x-interval $X_i$, some candidate $y$-interval is a good approximation of the interval $Y_i$ w.h.p., so the union of the LIS in all pseudo-solutions covers a large fraction of $\mathsf{OPT}$. Since there are $O(1/\lambda^3)$ pseudo-solutions, the output is a $O(1/\lambda^3)$-approximation of $\mathsf{OPT}$. The runtime of $\tilde{O}(\sqrt{n}/\lambda^7)$ is dominated by the time required to evaluate the quality of pseudo-solutions to sufficient accuracy. One can also apply this technique recursively to improve the runtime, at the cost of worse approximation.

In [34], this result is improved by giving an algorithm for LIS with approximation factor $O(1/\lambda^\epsilon)$ and runtime $O(n^{1-\Omega(\epsilon)} \cdot (\log n/\lambda)^{1/\epsilon})$ for any constant $\epsilon > 0$. The polynomial dependence on $1/\lambda$ seems essential since the algorithm in [42] is used as a sub-routine. In [37], a *non-adaptive* algorithm is presented with sample complexity $\tilde{O}(\sqrt{r}/\lambda^2)$ and approximation $O(1/\lambda)$, where $r$ is the number of distinct values. Again, the polynomial dependence of the sample complexity on $r$ and $1/\lambda$ seems intrinsic.

There are several fundamental obstacles in improving these bounds, and in particular, getting the runtime down all the way to $1/\lambda$ while improving the approximation. One such obstacle is that the straight-forward approach (as in prior work) of independent recursions cannot obtain better than $1/\lambda^2$ run-time in the worst-case. The issue is that even if one is able to isolate all possible $y$-interval ranges efficiently, there must still be at least $1/\lambda$ such ranges (optimally), and recusing of each such $y$-interval without knowing a priori which coordinates contain the relevant $y$-values would require $1/\lambda$ samples just to find a single sample of relevance in each (sub-)instance (i.e. for each $y$-interval). We will return to this obstacle later and show how we improve the *amortized complexity* for it.

### A. *Our approach*

Similarly to [42], we generate candidate $y$-intervals based on random samples, and generally look for increasing sequences of intervals, each with a large local LIS. Beyond this general similarity, our algorithm develops a few new ideas, in particular in how we work with these candidate intervals, and especially how we find LIS's among these candidate intervals. In particular, our contribution can be seen through three main components.

**First**, we introduce a new problem, termed $\mathsf{Genuine\text{-}LIS}$, which captures the problem of estimating the longest (sub-)sequence of increasing intervals, each with a large "local LIS". This problem is a new model for sublinear-time algorithms, which has not appeared in prior literature to the best of our knowledge.

**Definition III.1** ($\mathsf{Genuine\text{-}LIS}$)**.** *Given a sequence $g \in (\mathbb{N} \times \{0, 1\})^n$ where each element $g(i)_1 \in \mathbb{N}$ is associated* *with an additional flag $g(i)_2 \in \{0,1\}$ signifying whether it is* genuine *or not,* Genuine-LIS$(g)$ *is the length of the longest increasing subsequence of $g(*)_1$ restricted to genuine elements, i.e., of $g((g(*)_2)^{-1}(1))_1$.*

The Genuine-LIS problem can be described as follows. Given an input sequence $g \in (\mathbb{N} \times \{0,1\})^n$, one receives unrestricted access to the elements (i.e., the first coordinates) $g(*)_1 \in \mathbb{N}^n$. However, one must "pay" to test whether an element is genuine; this is determined by the second coordinates $g(*)_2 \in \{0,1\}^n$, referred to as genuineness flags. The goal is to compute Genuine-LIS(g) using as few *tests for genuineness* as possible.

**Second**, we show how to efficiently reduce the LIS problem to a (smaller) instance of the Genuine-LIS problem, where the "genuineness" flag of an element corresponds to LIS of an $x$-interval being large enough. In particular, this is where we generate the aforementioned candidate $y$-invervals. The algorithm reduces the problem to finding a (global) LIS amongst a set of candidate intervals, each with a large (local) LIS. We solve this via a composition of 2 sub-problems: the Genuine-LIS problem takes care of the "global LIS" of candidate-intervals, while the standard LIS problem recursively estimates the "local LIS" (these ideas and terms will be made precise later). The main challenge here is to generate candidate intervals in a *succinct* manner: we manage to reduce the number of sampled anchor needed to $\approx 1/\lambda$, as well as limiting the number of candidate intervals to be *near-linear* in the number of sampled points.

**Third**, to optimize the sampling of the string, we use a novel sampling scheme endowed with a data structure, termed *Precision-Tree*. At preprocessing, the structure samples a number of positions in the input string (non-adaptively), which are not uniform but rather correspond to a tree with non-uniform leaf levels (hence the name). We then build a global data structure on these samples for efficient access. In particular, we develop a *tree decomposition procedure* (to be discussed later), and repeatedly use *sub-trees* across different sub-instances we generate, improving the overall sample complexity. We highlight the fact that we *reuse* both the samples (and hence randomness) across different (sub-)instances for both of the LIS and Genuine-LIS problems. In addition, the Precision-Tree data structure provides a quick way to narrow down on all samples in some $x$-interval whose value is in given $y$-interval $Y$.

Before continuing this overview, we formally define a slightly generalized version of the LIS problem, termed Block-LIS, as well as the aforementioned Genuine-LIS problem. Following that, we provide an overview of the algorithms to solve these two problems. The high-level flow of our algorithm is described in Figure 1.

**The Block-LIS problem.** This problem extends the standard LIS problem in two ways. First, the main input is a sequence of $n$ blocks, each containing (at most) $k$ elements, and at most one element in each block may participate in a subsequence. Second, we are also given a range of values $Y \in \mathcal{I}$, such that each element of a subsequence must be in $Y$.

**Definition III.2** (Block-LIS)**.** *Given a sequence $y \in \mathbb{N}^{n\times k}$ and a range of values $Y \in \mathcal{I}$,* Block-LIS$(y, Y)$ *is the length of a maximal sub-sequence $OPT \subseteq [n] \times [k]$, such that $\{(w_1, y_w)\}_{w\in OPT}$ is monotone and $\{y_w\}_{w\in OPT} \subseteq Y$. Sometimes, we also restrict the set of blocks to an interval $X \subseteq [n]$, and define the quantity* Block-LIS$(y, X, Y)$ *as the longest increasing subsequence of $y \in \mathbb{N}^{X\times[k]}$ using elements in $Y$.*

The standard LIS problem can be seen as the Block-LIS problem with $k = 1$ and $Y = \mathbb{N}$ — which is how we instantiate the original input sequence $y \in \mathbb{N}^n$ in our Block-LIS algorithm. The main advantage of this generalization appears when the sequence is *sparse in* $Y$, i.e. most elements cannot participate in any LIS. Then, we show the multiplicative approximation factor for Block-LIS is not only a function of the additive error $\lambda n$, but also of the total number of elements of $y$ in range $Y$ (i.e., $|y(X, Y)|$). In particular, we show that this approximation is a function of $\frac{\lambda n}{|y(X,Y)|}$.

While the generalization to blocks is not a core necessity of the algorithm (in fact, one can consider the values in each block in descending manner, yielding an equivalent problem with no blocks required), its main use comes from our need to instantiate several overlapping instances using the same Precision-Tree data structure (to be discussed later).

Finally, we note that we similarly generalize the Genuine-LIS to work over blocks: input $g \in (\mathbb{N} \times \{0,1\})^{n\times k}$ and the LIS is allowed to use only one (genuine) element per block.

## B. Main Algorithm

The main algorithm for estimating the LIS is merely the following (see Algorithm 1):

1) Preprocess Precision-Tree $T = T_\delta(y)$ of the input sequence with parameter $\delta \triangleq \lambda/n^{o(1)}$ (the exact $o(1)$ parameter will be made precise later).
2) Run the Block-LIS algorithm that has access only to the data structure $T$. (We note that this algorithm itself recursively runs the algorithms for Genuine-LIS and Block-LIS.)

## C. *Genuine-LIS problem: overview of the algorithm* EstimateGenuineLIS

Recall that in the Genuine-LIS problem with input $g \in (\mathbb{N} \times \{0,1\})^n$, one receives unrestricted access to the elements (i.e., the first coordinates), but must "pay" to test whether an element is genuine (determined by the second coordinates). The goal is to compute the length of

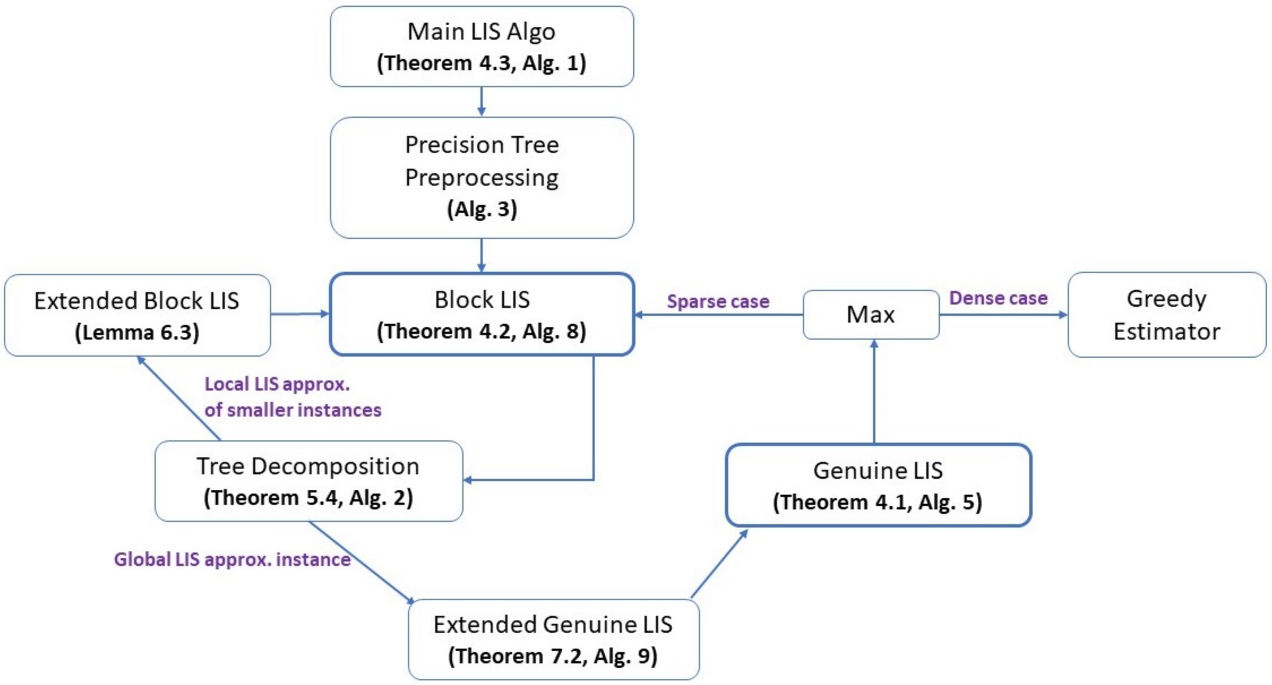

Fig. 1. High level flow of the main algorithm (Theorem I.1). Each arrow represents one or more calls to the corresponding algorithm. Detailed description and analysis of algorithms are in the full version of our paper.

the longest increasing subsequence of $g(*)_1$ restricted to genuine elements.

We derive two complementary solutions to the Genuine-LIS problem: one "direct" (without further recursion), and the other by reducing the problem to standard sublinear-time LIS estimation. To highlight our novel contribution, we contrast it with the framework of [42], that implicitly gives a solution to the Genuine-LIS problem, albeit with sub-optimal approximation. In particular, the following algorithm for Genuine-LIS is analogous to the algorithm in [42] based on generating pseudo-solutions (maximal sequences of candidate intervals).

1) Using the first coordinates only, look for a maximal *increasing subsequence* $P_1$ containing $\gtrapprox \lambda n$ elements and remove $P_1$ from the sequence.
2) Repeat step (1) until there are no more sequences of length $\gtrapprox \lambda n$. This generates (disjoint) solutions $P_1, \dots, P_t$ for some $t \lessapprox 1/\lambda$.
3) Sub-sample $\approx 1/\lambda$ elements from the union $P = \cup_{i \in t} P_i$ and check if each one is genuine. Let $\kappa$ be the number of Genuine elements in this sampled set.
4) Output $\lambda^2 n \cdot \kappa$.

It is easy to see that this algorithm yields, with constant probability, a $(2/\lambda, \lambda^2 n)$-approximation with approximately $1/\lambda$ tests for genuineness. First, if the Genuine-LIS is at most $2\lambda n$, then the output (being greater than 0) is already a $(2/\lambda, \lambda^2 n)$-approximation, as

$$\frac{\text{Genuine-LIS}}{2/\lambda} - \lambda^2 n \le \frac{2\lambda n}{2/\lambda} - \lambda^2 n = 0 \le \text{Genuine-LIS}.$$

Otherwise, $P$ contains most of the Genuine-LIS elements, and since $t \lessapprox 1/\lambda$, the Genuine-LIS is at least $\lambda/2$ times the number of genuine elements in $P$.

Thus, by the sampling mechanism above, we can estimate the proportion of genuine elements to sufficient accuracy with constant probability. Observe that the approximation is in fact proportional to $1/\lambda\kappa$, and that the worst approximation happens when $P$ is "sparse", i.e., when only approximately $\lambda n$ elements in $P$ are genuine. For such a *sparse case*, however, one can instantiate the Genuine-LIS algorithm as a Block-LIS one, restricted to the genuine elements. In fact, this "sparse" case is precisely where the approximation factor is minimal, i.e., $\frac{\lambda n}{|y|} = 1$ where $y$ is the genuine-elements-only subsequence. Our improved approximation stems, in part, from balancing between the dense and sparse cases as above.

**Speeding up LIS extraction using dynamic LIS.** The Genuine-LIS instances we generate are of size $\lessapprox 1/\lambda$, and our goal is to obtain overall run-time that is near-linear in $1/\lambda$ as well. The standard, dynamic-programming solution for finding and extracting the optimal LIS each time for step (1) above can potentially incur an overhead that is quadratic in the instance size (each LIS extraction would take linear time, and we need to greedily extract up to a near-linear number of solutions), leading to a polynomially-worse time. To ensure near-linear time, we adopt a fast dynamic LIS data structure from [23], and use it to iteratively extract near-maximal pseudo-solutions.

**Extensions to the Genuine-LIS algorithm.** Our complete algorithm needs two further extensions to the algorithm for the Genuine-LIS problem, as follows.

- *Sparse (unbalanced) instances:* some Genuine-LIS instances we generate are sparse, with many "null" elements, i.e., some blocks have less than $k$ elements (note that a "null" element is different from a "non-genuine" one, in that it does not need to be tested). When the instance $g$ consists of $m \ll nk$ non-null elements, we would like the approximation and runtime bounds to be a function of $m/n$ (average block size) instead of $k$ (maximum block size). To obtain the improved bound, we partition the blocks based on an exponential discretization of the number of non-null elements, and output the maximum over all instances, where each instance consists only of blocks containing a similar number of non-null elements.
- *Genuine-LIS over intervals:* the first coordinates of the Genuine-LIS instances we generate a priori consist of intervals rather than integers. The challenge is that we then need to find LIS over the *partial order* of intervals $\mathcal{I}$. To exemplify the challenge, the dynamic LIS data-structures from [23], [31], [33] which are useful in speeding up our algorithm, cannot immediately handle partial order sequences. Our ideal solution involves a map $\varphi : \mathcal{I} \to \mathbb{N}$ that approximately preserves the overall LIS over all subsequences, and therefore also preserves the overall LIS over the genuine elements. While we are unable to show a single mapping that works for all intervals, we partition the space of intervals into $\log(k/\lambda)$ sets $\mathcal{I}_\ell$ — intervals in $\mathcal{I}$ whose length is in $[\ell, 2\ell)$ — based on an exponential discretization of the interval length $\ell$, and provide a map $\varphi_\ell : \mathcal{I}_\ell \to \mathbb{N}$ for each set of intervals $\mathcal{I}_\ell$. We eventually output the maximal Genuine-LIS result over all such maps. This costs us merely another $\log(k/\lambda)$-factor approximation, and a small additive error.

The formal statement for the algorithm to solve Genuine-LIS, named EstimateGenuineLIS, is presented in Section IV. The algorithm description and analysis, as well as its extensions, are postponed to the full version [8].

## D. *Block-LIS problem: overview of the algorithm* EstimateBlockLIS

For the problem Block-LIS$(y, X, Y)$, we develop the algorithm EstimateBlockLIS. Fundamentally, we reduce the instance to a Genuine-LIS instance, where each genuineness flag is set to 1 if and only if the LIS of a certain (smaller) sequence (which is itself a Block-LIS problem) is large enough.

Our reduction starts by partitioning $X$ into consecutive intervals $X_1, \dots, X_\tau$ of equal length, where $\tau$ is a dynamic, carefully chosen branching factor, and is usually sub-polynomial in the instance size (here think $\tau = n^\epsilon$). Next, we sample approximately $\tau/\lambda$ blocks $(w_j, y_{w_j})$ (termed *anchors*), and use all the elements in $Y$ across all the anchors to generate sets of $y$-values $\mathcal{S}_i \subseteq Y$ for each $X_i$.

Using each set of $y$-values $\mathcal{S}_i$, we construct *candidate intervals* $\mathcal{Y}_i$. While this part is similar to the construction from [42], we note that we need a more efficient construction to obtain the near-optimal sample complexity. In particular, we require the number of candidate intervals to be near-linear in $|\mathcal{S}_i|$, and hence we construct a *small* candidate interval set $|\mathcal{Y}_i| \approx |\mathcal{S}_i|$, while still ensuring that this set covers all "relevant options" with only an extra logarithmic factor loss in approximation. In particular, instead of looking for all possible pairs of endpoints, we choose the endpoints in dyadic fashion (in particular, their distance in the set $\mathcal{S}$ must be a power of 2.

Finally, we reduce the Block-LIS problem to a Genuine-LIS instance over $\tau$ blocks, where each block $i$ contains all the candidates intevals $\mathcal{Y}_i$ — this Genuine-LIS instance thus captures "global" LIS (over the candidate intervals). The first coordinates of the Genuine-LIS instance are the candidate intervals $\mathcal{Y}_i$ themselves, while the second coordinates (i.e., the genuineness flags) indicate whether the corresponding "local" Block-LIS$(y, X_i, Y')$ estimated values are above a certain threshold $\kappa$, for each $Y' \in \mathcal{Y}_i$.

This threshold $\kappa$ itself depends on a parameter $\rho > 1$, which characterizes the relation between the "global" Genuine-LIS instance and the "local" Block-LIS instances. Intuitively, $\rho$ is such that (roughly) $1/\rho$ fraction of the $X_i$ intervals each have a $\lambda\rho$ fraction of them participating in the LIS. Consider two extreme cases, one where the LIS is uniformly distributed among all $X_i$ ($\rho = 1$), and one where the LIS is maximally concentrated among a small subset of the $x$-intervals ($\rho = 1/\lambda$). Then, it is more difficult to certify an increasing subsequence in the latter case, when the LIS is sparse. But in this case testing the genuineness of a local LIS (inside a block $X_i$) is much easier, since we only need to establish that it is $\Omega(|X_i|)$. In general, there is a precise trade-off between the complexity of certifying the global LIS versus the complexity of certifying the local Block-LIS. A priori, we do not know $\rho$, so we simply iterate over all possible magnitudes (again, by exponential discretization).

Once we formulate the overall Block-LIS problem as a composition of a "global" Genuine-LIS over multiple "local" Block-LIS instances, we decompose the problem through a procedure called *Precision-Tree decomposition* which will be described next.

The formal statement for the EstimateBlockLIS algorithm is presented in Section IV. The algorithm description and analysis are postponed to the full version [8].

## E. Precision-Tree data structure

We note that a direct instantiation of the above algorithm yields the right approximation, but will require at least $\approx 1/\lambda^2$ queries into the input string $y$ (and hence run-time), and, moreover, require the algorithm to be *adaptive*. The source of this inefficiency is precisely the "small slopes" obstacle discussed earlier, which would lead to "wasting samples" in any naïve rejection sampling mechanism. To improve the complexity, and to allow our sampling algorithm to be non-adaptive, we introduce the *Precision-Tree* data structure (Section V). While based on the *precision sampling* tool introduced in [5], [6], the main development here is that we augment the resulting tree for special operations described henceforth.

Intuitively, the Precision-Tree data structure can be thought of as a global data structure holding enough information for simulating random samples for multiple non-uniform (sub-)instances, specifically ones arising from recursion. In particular, the data structure is an (incomplete) tree, where (lowest-level) leafs correspond to samples of some input $e \in \mathcal{E}^n$. The initial tree is defined over the input string (i.e., $\mathcal{E} = \mathbb{N}$), but we also construct Precision-Trees over other domains $\mathcal{E}^n$ as described later. The initial tree for the string $y \in \mathbb{N}^n$ is constructed at random (using precision sampling), and this is the only time when we access the string $y$ (in the entire LIS algorithm). Overall, the Precision-Tree data-structure (the initial one as well as the other trees we generate) supports several important properties we need:

- Since our algorithm is recursive (composition of Genuine-LIS and Block-LIS algorithms), we need to provide samples for potentially very small parts of $y$ (that depend on what was sampled earlier in the recursion). In other words, we need to be able to "zoom into" different location of the input string with the right precision.
  Our precision tree is build such that any sub-tree under any "sampled" node, is another Precision-Tree instantiation with a different (random) parameter.
- In contrast to the original precision sampling tool that was designed for the simple addition function, our application requires more complicated functions (the sub-linear Block-LIS and Genuine-LIS algorithms).
  We show how we can *decompose* the tree into a number of different sub-trees (as mentioned, each sub-tree by itself is a precision tree), run algorithms on each sub-tree as well as a final algorithm, recomposing the results using another algorithm (and another sub-tree, this time over a new domain resulting from the sub-trees computation).
- Another crucial challenge to deal with is the following: as the recursion proceeds, we may be "zooming in" on the same part of $y$ multiple times, but focusing on different $y$-range intervals $Y$. Naïvely, one would use rejection sampling here (rejecting samples with value not in $Y$), which however would yield a polynomially worse sample complexity. In addition, our recursion happens not only down the tree (i.e., on sub-trees) but also on the top portion of the tree (reminiscent of the van Emde Boas layout).
  The precision tree allows us to *reuse* the randomness over independent recursive calls into the same $x$-intervals, each focusing on a different $y$-value ranges. In particular, for such an $x$-interval we will have processed the samples to directly report samples with values in a desired $y$-interval $Y$ (this is essentially a 2D range reporting). In addition, the randomness is reused when computing the global function on the top portion of the tree as described above.

*1) Construction of the Precision-Tree data structure:* We now describe the construction in a bit more detail. Original precision sampling from [5], [6] is designed to estimate a *summation* function $a = \sum_{i=1}^n a_i$, for unknown $a_i \in [0, 1]$, from "coarse" estimates for $a_i$. In our application, we need to *generalize* precision sampling from a simple addition to *general functions*, allowing one to approximate $g \circ f$ where $g$ is a general function over $m$ coordinates and $f = (f_1, \dots, f_n)$ consists of $n$ *independent* functions on different parts of the input, sharing the same co-domain.

We define Precision-Tree with precision parameter $\lambda < 1$, for a given a vector of elements $e = (e_1, \dots, e_n) \in \mathcal{E}^n$, denoted $T_\lambda(e)$, as follows. The tree $T$ is a trimmed version of the full $\beta$-ary tree of $\ell = \log_\beta n + 1$ levels, with $n = \beta^\ell$ leafs, where parameter $\beta > 1$ is fixed; in particular, all nodes of $T$ have fan-out of either $\beta$ or 0. Each leaf at level $\ell$ [2] is associated with an integer representing its location in the tree. Each internal node $v$ is associated with an interval representing the unions of its leaf descendants (in the full tree); for example, the root of $T$ is associated with the entire interval $[1, n]$, and its children with $[1, n/\beta]$, $[n/\beta + 1, 2n/\beta]$, etc). For a node $v$ in the tree, we use $\mathcal{E}_v$ or $\mathcal{E}(v)$ to denote the set of elements under $v$.

Given $e \in \mathcal{E}^n, \lambda$, and $\beta$, we construct the trimmed Precision-Tree $T_\lambda(e)$ as follows. We assign a (random) precision score $\mathcal{P}_v$ to each node $v$ by the following recursive procedure. We set $\mathcal{P}_{\text{root}(T)} = \lambda$. Recursively we define the precision score of a node $v$ by

$$\mathcal{P}_v = \mathbf{Z}_v \cdot \mathcal{P}_{\text{parent}(v)} \text{ where } \mathbf{Z}_v \sim_{i.i.d.} \texttt{Uniform}(\{1, 2, 3, \dots, \beta/4\}). \quad (1)$$

Most importantly, for a node $v$, we recurse into its children only if $\mathcal{P}_v \leq 1$; otherwise we stop the recursion for $v$ with $\mathcal{P}_v > 1$ ($v$ has 0 children). If $v$ is a leaf (at level $\ell$), $v$ also stores the input element corresponding to the location of $v$ (i.e., an element of $e$) — note that this event corresponds to sampling an element of the input string. So, we store only a "trimmed" version of a full $\beta$-ary tree.

[2]By convention, the level of the root is 0.

Initially, we generate one such tree $T_{1/c}(y)$ for the input string $y \in \mathbb{N}^n$, for some $c = c(n, \lambda) \geq 1$, which we refer to as "precision tree complexity". One can prove that the sample complexity (into $y$) is bounded by $c \cdot \log^{O(\log_\beta n)}(\beta)$ (see Lemma V.1). Overall, our entire sampling mechanism is merely a simple, one-round non-adaptive precision-sampling tree over the coordinates of $y$, which we store in a data structure $T$[3] of size $O^*(1/\lambda)$ with convenient fast access.[4] After such preprocessing, the rest of the algorithm has no access to $y$, but only to $T$.

Most importantly, using this single tree $T_{1/c}(y)$, we will simulate Precision-Tree access for different trees over different inputs and parameters in our algorithms using a certain procedure which we call the Trim-Tree algorithm (see Algorithm TRIMTREE in the full version [8]).

We also preprocess the samples stored in $T$ so that we can quickly retrieve all samples in some $x$-interval $X \subseteq [n]$ and some $y$-interval $Y$ through an ancillary data-structure which we call SODS. The SODS is used for a "rejection sampling"-type operations where we need a number of samples from a substring of input $y$ (or later on for some intervals of blocks), but care only about values in a certain range $Y$.

## IV. Main Algorithm for Estimating LIS

Our main algorithm is composed of two algorithms, for solving Block-LIS and Genuine-LIS. These algorithms recursively call each other, with access to a Precision-Tree data structure. This data structure queries the input sequence at the beginning (non-adaptively and non-uniformly), and our algorithms access the sequence only via this data structure. We describe the details of Precision-Tree in Section V, and for now refer to it as a tree $T_\lambda(y)$ for some parameter $\lambda < 1$ and an input string $y$.

We now state the main guarantees of the two algorithms; their proof will appear in later sections. The statements below are built to support mutual recursion and might be challenging to parse at first read. At high-level, the idea is to create mutual inductive hypotheses which can be instantiated with different parameters, in a way that enables concurrent optimization of the approximation, sample complexity and run-time complexity. We then show how careful recursive instantiations of these two algorithms yield our main algorithm for estimating LIS.

Below, for an instance $g \in (\mathbb{N}\times\{0,1\})^{n\times k}$, let $g$ restricted to first/second coordinate be $g(*)_1 \in \mathbb{N}^{n\times k}$ and $g(*)_2 \in \{0,1\}^{n\times k}$ respectively.

**Theorem IV.1** (EstimateGenuineLIS algorithm)**.** *Fix integers $n, k$ and $\lambda \in (1/n, 1)$. Fix an instance $g \in (\mathbb{N} \times \{0,1\})^{n\times k}$. For some monotone functions $a_s : \mathbb{R}^3_+ \to [1,\infty)$ and $c_s : \mathbb{R}^4_+ \to [1,\infty)$. Suppose there exists a randomized algorithm $\mathcal{A}_{BL}$ that, given $X \subseteq [n], Y \subseteq \mathbb{N}$, parameter $\tau_s$, and a Precision-Tree $T_{1/c_s\left(|X|,\tau_s,\lambda,\frac{\lambda\cdot|X|}{|y(X,Y)|}\right)}(y)$ for some $y \in \mathbb{N}^{n\times k}$, can produce an $(\alpha_s, \lambda|X|)$-approximation for Block-LIS$(y, X, Y)$ where $\alpha_s = a_s\left(|X|, \tau_s, \frac{\lambda\cdot|X|}{|y(X,Y)|}\right)$ w.h.p. in time $t_s$.*

*Fix any parameter $\gamma, \tau \geq 1$, and let $c \geq \tilde{O}(1/\lambda) + c_s(n, \tau, \lambda, \gamma\lambda/k)$. Then, there exists an algorithm $\mathcal{A}$, that, given free access to $g(*)_1$ and Precision-Tree access to $g(*)_2$, $T_{1/c}(g(*)_2)$, produces a $(\alpha_g, \lambda n)$-approximation for Genuine-LIS$(g)$ w.h.p., where $\alpha_g = \log(k/\lambda) \cdot \max\{\gamma, a_s(n, \tau, \gamma\lambda/k)\}$. The algorithm $\mathcal{A}$ runs in time $\tilde{O}(nk) + O(t_s)$.*

**Theorem IV.2** (EstimateBlockLIS algorithm)**.** *Fix monotone functions $t_s, a_g : \mathbb{R}^2_+ \to [1,\infty)$, $a_s, c_g : \mathbb{R}^3_+ \to [1,\infty)$ and $c_s : \mathbb{R}^4_+ \to [1,\infty)$, satisfying, for all $r, \tau \in E_\beta$, $m \in \mathbb{N}$, $\lambda < 1$, $\lambda_1, \lambda_2 \in [\lambda, 1]$, and $k' \in [1, 1/\lambda]$ with $\lambda_1\lambda_2 = \Omega(\lambda k')$:*

- $a_s\left(r, \tau, \frac{\lambda}{m}\right) \geq \operatorname{polylog}\left(\frac{k}{\lambda}\right) \cdot a_g\left(\tau, \frac{\lambda_2}{k'}\right) \cdot a_s\left(\frac{r}{\tau}, \tau, \frac{\lambda_1}{m}\right)$;
- $c_s\left(r, \tau, \lambda, \frac{\lambda}{m}\right) \geq \beta^{O(1)} \cdot \frac{\tau}{\lambda} + c_g(\tau, \lambda_2, k') \cdot c_s\left(\frac{r}{\tau}, \tau, \Theta(\lambda_1/k'), \frac{\lambda_1}{m}\right)$*; and,*
- $t_s\left(r, \tau\right) \geq \log^{O(\log_\beta(\tau))}(\beta) \cdot t_s\left(\frac{r}{\tau}, \tau\right)$.

*Suppose there exists and algorithm $\mathcal{A}_{GL}$ with the following guarantee: given Genuine-LIS instance $g \in (\mathbb{N} \times \{0,1\})^{n_g\times k_g}$ with $\beta$-ary Precision-Tree $T_{1/c_g(n_g,\lambda_g,k_g)}(g(*)_2)$ access, $\mathcal{A}_{GL}$ outputs $\left(a_g\left(n_g, \frac{\lambda_g}{k_g}\right), \lambda_g n_g\right)$-approximation to Genuine-LIS$(g)$ in time $\tilde{O}(n_g \cdot k_g)$ w.h.p.*

*Now fix input $y \in \mathbb{N}^{n\times k}$, a block interval $X \subseteq [n]$, value range interval $Y \subseteq \mathbb{N}$, parameters $\lambda \in (0,1)$, $\beta \in \mathbb{N}$, $\tau \in E_\beta$. Then, given a $\beta$-ary Precision-Tree $T_{1/c}(y)$, we can produce a $(\alpha, \lambda|X|)$-approximation for Block-LIS$(y, X, Y)$ w.h.p., as long as $\alpha \geq a_s\left(|X|, \tau, \frac{\lambda|X|}{|y(X,Y)|}\right)$ and $c \geq c_s\left(|X|, \tau, \lambda, \frac{\lambda|X|}{|y(X,Y)|}\right)$. The algorithm's expected run-time is at most $c \cdot t_s(|X|, \tau) \cdot \frac{|y(X,Y)|}{|X|}$.*

The proofs are deferred to the full version [8]. Combining the two algorithms above, we obtain the following theorem.

**Theorem IV.3.** *Fix any $\lambda = o(1)$ and $\epsilon < 1$. There exists a randomized non-adaptive algorithm that solves the LIS problem with $(\alpha, \lambda n)$-approximation, where $\alpha = (1/\lambda)^{\sqrt{\epsilon}} \cdot (\log 1/\lambda)^{2^{O(\log^2(1/\epsilon)/\sqrt{\epsilon})}}$ using $\frac{1}{\lambda} \cdot n^{O(\sqrt{\epsilon}\log 1/\epsilon)}$ time (and hence samples from the input).*

The algorithm for Theorem IV.3 (see Algorithm 1) follows the outline in Figure 1. We first build a $\beta$-ary precision tree $T_{1/c}$ for $\beta = \Theta(\log n)$, and $c = c_s(n, n^\epsilon, \lambda, \lambda) = \frac{1}{\lambda} \cdot n^{O(\sqrt{\epsilon}\log 1/\epsilon)}$. We then apply the algorithm of Theorem IV.2 by interpreting the string as a Block-LIS instance with $k = 1$. The proof follows from recursive implementation of Theorems IV.1 and IV.2 with carefully chosen parameters. Most importantly $\tau$ and $\gamma$ are carefully chosen, as a function

[3]We abuse notation and use $T$ to denote both the conceptual Precision-Tree and the data structure implementing it.

[4]The notation $O^*(\cdot)$ hides a $n^{o(1)}$ factor.

of the other parameters, to balance approximation and complexity. Informally, we pick $\tau \approx |X|^{\epsilon}$ in each Block-LIS instance, and $\gamma \approx (k/\lambda)^{\sqrt{\epsilon}}$ for a recursion of depth $\approx 1/\sqrt{\epsilon}$, then we stop and use the dense estimator only by setting $\gamma$ to be maximal. The formal proof involves tedious calculations, and is deferred to the appendix in the full version [8]. We now complete Theorem I.1, by instantiating the LIS problem with the parameters above set suitably.

```
Algorithm 1: EstimateLISMain
  Input: A sequence $y \in \mathbb{N}^n$, error parameter
         $\lambda \in (1/n, 1)$, and $\epsilon < 1$.
  Output: An integer $\widehat{L} \in [0, \mathrm{LIS}(y)]$.
1 $T \leftarrow$ PreprocessPrecisionTree($y, \lambda \cdot n^{-\epsilon}, \beta$),
    where $\beta \leftarrow \Theta(\log n)$.
2 return $\widehat{L} \leftarrow$ EstimateBlockLIS($T, n^{\epsilon}, \lambda$).
```

Fig. 2. Description of the algorithm EstimateLISMain.

*Proof of Theorem I.1.* Let $y \in \mathbb{N}^n$ be an input of the LIS problem. We solve the problem using the algorithm of Theorem IV.3 with inputs $y$, $\lambda$, and $\epsilon = 1/\log\log 1/\lambda$, noting that $\alpha = \lambda^{o(1)}$ and runtime complexity is $\frac{1}{\lambda} n^{O(1/\sqrt[3]{\log\log 1/\lambda})} = \frac{1}{\lambda} n^{o(1)}$. Finally, we invoke Fact II.3 with $q = o(1)$ to obtain the claimed approximation (as $q = o(1)$, $p = q/(1-q) = o(1)$). $\square$

## V. Precision-Tree Data Structure

In this section we discuss the Precision-Tree data structure used to improve our bounds, and its properties.

The basic construction of the $\beta$-ary tree $T_\lambda(e)$ was introduced in Section III-E1. Before proceeding, we establish the sample complexity of the Precision-Tree. In the below, we use $V_\ell$ to denote the set of nodes at level $\ell$.

**Lemma V.1** (Sample complexity)**.** *The expected number of coordinates in $[n]$ that are sampled is $\log^{O(\log_\beta n)}(\beta)/\lambda$.*

*Proof.* Let $s_v = 1/\mathcal{P}_v$. Since we only sample leaves with $s_v \geq 1$, then the total sampled leaves is at most $\sum_{v \in V_{\log_\beta(n)}} s_v$.

Now, since $\mathbf{Z}_v$ are chosen i.i.d. uniformly in $\{1, \ldots, \Omega(\beta)\}$, then we deduce the recursive relation:

$$\mathbb{E}[s_v] \leq \mathbb{E}\left[\frac{1}{\mathcal{P}_v}\right] = \frac{O(\log\beta)}{\beta} \cdot \mathbb{E}\left[\frac{1}{\mathcal{P}_{\mathrm{parent}(v)}}\right] \tag{2}$$

Let $\tau_\ell \triangleq \sum_{v \in V_\ell} s_v$. Then $\tau_0 = 1/\lambda$, and for level $\ell$, summing and using Equation (2), we obtain

$$\mathbb{E}[\tau_\ell] = \mathbb{E}\left[\sum_{v \in V_\ell} s_v\right] \leq \beta \cdot \mathbb{E}\left[\sum_{v \in V_{\ell-1}} \frac{O(\log\beta)}{\beta} \cdot s_v\right] \leq O(\log\beta)\, \mathbb{E}\left[\sum_{v \in V_{\ell-1}} s_v\right] = O(\log\beta)\tau_{\ell-1}.$$

Then $\mathbb{E}[\tau_\ell] \leq O(\log\beta)^\ell/\lambda$, and we conclude $\mathbb{E}\left[\sum_{v \in V_{\log_\beta(n)}} s_v\right] = O(\log(\beta))^{\log_\beta n}/\lambda$ as needed. $\square$

### A. Tree Sampling Data Structure

We equip each Precision-Tree with a *Sampling Oracle Data Structure* (denoted SODS) that should be seen as a data structure wrapper with access to Precision-Tree, allowing *efficient sampling* of leaves.

First, we show one can simulate uniform samples of elements given a Precision-Tree access.

**Lemma V.2** (Simulating Random Samples)**.** *Fix $\delta \leq 1$ and Precision-Tree $T = T_\delta(e_1, \ldots, e_n)$, as well as arbitrary $\delta' \geq \delta$, with $1/\delta' \in \mathbb{N}$. Using access to $T$ only, we can generate a set of elements $S \subseteq \{e_1, \ldots, e_n\}$ such that the distribution of $S$ is identical to the distribution where each $e_i$ is included i.i.d. with probability $1/\delta' n$. The runtime to generate $S$ is $O((\log\beta)^{\log_\beta n}/\delta)$.*

We remark that, while two subsequent invocations of the above lemma may give different sets $S, S'$, each with the above distribution, they are dependent between each other.

*Proof.* The main task here is to show independence, i.e., that we can choose a set of i.i.d. samples by reusing the randomness of the Precision-Tree. We prove by induction on the tree height, starting with single node trees, that for any node with $\mathcal{P}_v \leq 1$, for any $\delta' \geq \mathcal{P}_v$ we can generate a subset of the leaves where each leaf is included with probability at least $1/\delta' n_v$, where $n_v$ is the number of leaves in the subtree rooted at $v$.

For leaves, if $\mathcal{P}_v \leq 1$, the node is included in the Precision-Tree, and we can subsample it with any required probability as needed. Now consider a non-leaf node $v$; by induction the statement holds for its children.

If $\delta' \leq \frac{4}{\beta}$, we are basically done since any child $u$ of $v$, has score $\mathcal{P}_u \leq \frac{\mathcal{P}_v \beta}{4} \leq \frac{\delta' \beta}{4} \leq 1$, while the number of elements $n_u = \frac{n_v}{\beta}$, so we can simply subsample leaves in $u$'s tree, each with probability $\frac{1}{\delta' n_v} = \frac{1}{\delta' \beta n_u} \leq \frac{1}{\mathcal{P}_u n_u}$ as needed (note that $\delta' n_v / n_u = \beta\delta' \geq \mathcal{P}_u$ as required by the inductive hypothesis).

It remains to show this for $\delta' > \frac{4}{\beta}$. For this case we claim we would like to use the tree-randomness to generate the samples.

For distributions $P, Q$, we say that $P$ *stochastically dominates* $Q$ (also denoted $P \succ Q$) if $\mathcal{F}_P(t) \leq \mathcal{F}_Q(t)$ for all $t$, with strict inequality for some $t$, where $\mathcal{F}_P$ and $\mathcal{F}_Q$ are the cumulative distribution functions (CDFs) of $P$ and $Q$ respectively. We first claim the following:

**Claim V.3.** *Fix $k \in \mathbb{N}$, $0 < p < 1$ and $F \geq 1$ such that $F \cdot pk < 1/4$. Let $\mathbf{X}_1, \ldots, \mathbf{X}_k \sim_{i.i.d.} \texttt{Ber}(p)$ (i.e., i.i.d. Bernoulli random variables with bias $p$) and let $\mathbf{X} = \sum_i \mathbf{X}_i$. Let $\mathbf{Y}_1, \ldots, \mathbf{Y}_k \sim_{i.i.d.} \texttt{Ber}(2\,p\,F)$ and let $\mathbf{Y} = \mathbf{Y}^* \cdot \sum_i \mathbf{Y}_i$ where $\mathbf{Y}^* \sim \texttt{Ber}(1/F)$, independent of the other variables. Then, $\mathbf{Y}$ stochastically dominates $\mathbf{X}$.*

The proof is deferred to the full version of our paper [8].

We show that one can simulate sub-sampling of each leaf of $u$ i.i.d. with probability $1/\delta' n_v$ by the following process. If $\mathcal{P}_u > 1$, then output no elements. Otherwise, we can (by the inductive hypothesis) sub-sample each leaf of $u$ i.i.d. with probability $1/n_u$.

Now, let

$$F \triangleq \frac{1}{\lceil 1/\delta' \rceil} \frac{\beta}{4} \geq \frac{1}{\lceil 1/\mathcal{P}_v \rceil} \frac{\beta}{4} \overset{Eq.\ 1}{\geq} \frac{1}{\Pr_T[\mathcal{P}_u \leq 1]}.$$

This process generates $\mu$ i.i.d. leaves of $u$ with distribution

$$\mu \sim \texttt{Ber}(\Pr_T[\mathcal{P}_u \leq 1]) \cdot \sum_{l \text{ leaf of } u} \texttt{Ber}(\tfrac{1}{n_u})$$
$$\succ \texttt{Ber}(\tfrac{1}{F}) \cdot \sum_{l \text{ leaf of } u} \texttt{Ber}(\tfrac{1}{n_u}).$$

Noting that $n_u = \frac{n_v}{\beta} \geq \frac{\delta' n_v}{2F}$ and $\frac{n_u F}{\delta' n_v} \leq \frac{1}{4}$, we invoke Claim V.3 to obtain that the distribution $\mu$ *stochastically dominates* the required sample-size distribution $\sum_{l \text{ leaf of } u} \texttt{Ber}(\frac{1}{\delta' n_v})$, and hence we can simulate the distribution we need over leaves of each child, and hence also for $v$.

For run-time, we note that this process takes time (at most) proportional to the size of the tree. For the latter, by Lemma V.1, the time is $O((\log \beta)^{\log_\beta n}/\delta)$. □

Next, we show that if each element of the tree is a block of integers, then one can construct a data structure that sub-samples with *interval range restriction*, in time proportional to the sample size.

**Corollary V.4** (Conditional sub-sampling data structure)**.** *Fix Precision-Tree $T = T_\delta(y)$ with $y \in \mathbb{N}^{n \times k}$. There exists a data structure, that given any interval $Y \in \mathcal{I}$ and sub-sampling probability $\eta \leq 1/\delta n$ with $1/\eta \in E_2(n)$, sub-sample block-coordinates $X' \subseteq [n]$ i.i.d. with probability $\eta$ and outputs a set of all coordinates $W \subseteq X' \times [k]$ such that $y_W \in Y$, i.e. $W \triangleq \cup_{i \in X'} \{(i,j) \mid y_{i,j} \in Y\}$, in time $\tilde{O}(|W|)$. The preprocessing time is, in expectation, $\tilde{O}\left(\frac{n + \|m\|_1}{\delta n}\right)$, where $m_i \leq k$ is the number of non-null elements in $y_i$.*

*Proof.* To preprocess, we prepare for each possible $\eta \leq 1/\delta n$ with $1/\eta \in E_2(n)$ by sub-sampling each block i.i.d. with probability $\eta$ using Lemma V.2. Then we compute a set of coordinates $Z_\eta \subset [n]$ by combining all coordinates from all sampled blocks. Let $U_\eta \triangleq \cup_{w \in Z_\eta \times [k]} \{y_w\}$. Compute $U_\eta$ and store it as a sorted array with a pointer to $y^{-1}(j) \cap (Z_\eta \times [k])$ for each $j \in U_\eta$. Now, for each $(\eta, Y)$ query, locate the range of elements in $U_\eta$ containing precisely the elements in $Y$ using binary search of $\min(Y), \max(Y)$. This will be our output $W$.

The correctness follows immediately from the construction. Preprocessing runtime is as claimed as, in expectation, we need to sort $\|m\|_1/\delta n$ elements, and there are $O(\log n)$ different values of $\eta$ to consider. Runtime is only $O(\log n)$ plus the output size (as it is stored as contiguous block). □

We can show that if one can compute some "local" function over intervals of elements given some local Precision-Tree parameter access to each interval, then we can redefine the tree-access as a tree-access to the global problem assuming all local problems were computed successfully. We defer the discussion on Precision-Tree decomposition to achieve this goal to the full version [8].

## References

[1] N. Ailon, B. Chazelle, S. Comandur, and D. Liu, "Estimating the distance to a monotone function," *Random Structures and Algorithms*, vol. 31, pp. 371–383, 2007, previously in RANDOM'04. I, I-B

[2] D. Aldous and P. Diaconis, "Longest increasing subsequences: from patience sorting to the baik–deift–johansson theorem," *Bulletin of the American Mathematical Society*, vol. 36, no. 04, pp. 413–432, 1999. I

[3] A. Andoni, P. Indyk, and R. Krauthgamer, "Overcoming the $\ell_1$ non-embeddability barrier: Algorithms for product metrics," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2009, pp. 865–874. I-B

[4] A. Andoni and R. Krauthgamer, "The smoothed complexity of edit distance," *ACM Transactions on Algorithms*, vol. 8, no. 4, p. 44, 2012, previously in ICALP'08. [Online]. Available: http://doi.acm.org/10.1145/2344422.2344434 I-B

[5] A. Andoni, R. Krauthgamer, and K. Onak, "Polylogarithmic approximation for edit distance and the asymmetric query complexity," in *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, 2010, full version at http://arxiv.org/abs/1005.4033. III-E, III-E1

[6] ——, "Streaming algorithms from precision sampling," in *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, 2011, full version at http://arxiv.org/abs/1011.1263. III-E, III-E1

[7] A. Andoni and H. L. Nguyen, "Near-tight bounds for testing Ulam distance," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010. I

[8] A. Andoni, N. S. Nosatzki, S. Sinha, and C. Stein, "Estimating the longest increasing subsequence in nearly optimal time," *CoRR*, vol. abs/2112.05106, 2021. [Online]. Available: https://arxiv.org/abs/2112.05106 I-A, I-A, III-C, III-D, III-E1, IV, IV, V-A, V-A

[9] O. Ben-Eliezer, C. Canonne, S. Letzter, and E. Waingarten, "Finding monotone patterns in sublinear time," in *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, 2019, pp. 1469–1494. I

[10] H. Black, D. Chakrabarty, and C. Seshadhri, "Domain reduction for monotonicity testing: A o(d) tester for boolean functions in d-dimensions," in *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '20. USA: Society for Industrial and Applied Mathematics, 2020, p. 1975–1994. I

[11] M. Boroujeni and S. Seddighin, "Improved mpc algorithms for edit distance and ulam distance," in *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '19. New York, NY, USA: Association for Computing

Machinery, 2019, p. 31–40. [Online]. Available: https://doi.org/10.1145/3323165.3323205 I

[12] K. Bringmann and D. Das, "A Linear-Time $n^{0.4}$-Approximation for Longest Common Subsequence," in *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), N. Bansal, E. Merelli, and J. Worrell, Eds., vol. 198. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 39:1–39:20. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/14108 I

[13] A. Chakrabarti, "A note on randomized streaming space bounds for the longest increasing subsequence problem," *Inf. Process. Lett.*, vol. 112, no. 7, p. 261–263, Mar. 2012. [Online]. Available: https://doi.org/10.1016/j.ipl.2011.12.008 I-B

[14] D. Chakrabarty and C. Seshadhri, "An optimal lower bound for monotonicity testing over hypergrids," *Theory of Computing*, vol. 10, no. 17, pp. 453–464, 2014. [Online]. Available: http://www.theoryofcomputing.org/articles/v010a017 I

[15] D. Das and B. Saha, "Approximating lcs and alignment distance over multiple sequences," 2021. [Online]. Available: https://arxiv.org/abs/2110.12402 I

[16] Y. Dodis, O. Goldreich, E. Lehman, S. Raskhodnikova, D. Ron, and A. Samorodnitsky, "Improved testing algorithms for monotonicity," *Electron. Colloquium Comput. Complex.*, vol. 6, no. 17, 1999. I

[17] F. Ergün and H. Jowhari, "On distance to monotonicity and longest increasing subsequence of a data stream," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2008, pp. 730–736. I-A, I-B, I-B, I-B

[18] F. Ergün, S. Kannan, R. Kumar, R. Rubinfeld, and M. Viswanathan, "Spot-checkers," *J. Comput. Syst. Sci.*, vol. 60(3), pp. 717–751, 2000. I

[19] E. Fischer, "The art of uninformed decisions: A primer to property testing," *Science*, vol. 75, pp. 97–126, 2001. I

[20] ——, "On the strength of comparisons in property testing," *Inf. Comput.*, vol. 189, no. 1, p. 107–116, Feb. 2004. [Online]. Available: https://doi.org/10.1016/j.ic.2003.09.003 I

[21] A. Gál and P. Gopalan, "Lower bounds on streaming algorithms for approximating the length of the longest increasing subsequence," in *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, 2007, pp. 294–304. I-B

[22] A. Ganesh, T. Kociumaka, A. Lincoln, and B. Saha, "How compression and approximation affect efficiency in string distance measures," 2021. [Online]. Available: https://arxiv.org/abs/2112.05836 I

[23] P. Gawrychowski and W. Janczewski, "Fully dynamic approximation of lis in polylogarithmic time," in *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 654–667. [Online]. Available: https://doi.org/10.1145/3406325.3451137 I-B, III-C, III-C

[24] ——, "Conditional lower bounds for variants of dynamic lis," 2021. I-B

[25] O. Goldreich, *Introduction to property testing*. Cambridge University Press, 2017. I

[26] P. Gopalan, T. S. Jayram, R. Krauthgamer, and R. Kumar, "Estimating the sortedness of a data stream," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007, pp. 318–327. I-A, I-B, I-B

[27] D. Gusfield, *Algorithms on strings, trees, and sequences*. Cambridge: Cambridge University Press, 1997. I

[28] M. Hajiaghayi, M. Seddighin, S. Seddighin, and X. Sun, *Approximating LCS in Linear Time: Beating the $\sqrt{n}$ Barrier*, 2019, pp. 1181–1200. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611975482.72 I

[29] J. Hammersley, "A few seedlings of research," *Proc. Sixth Berkeley Symp. Math. Statist. and Probability*, pp. 345–394, 1972. I

[30] S. Im, B. Moseley, and X. Sun, "Efficient massively parallel methods for dynamic programming," in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 798–811. [Online]. Available: https://doi.org/10.1145/3055399.3055460 I-B

[31] T. Kociumaka and S. Seddighin, "Improved dynamic algorithms for longest increasing subsequence," in *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 640–653. [Online]. Available: https://doi.org/10.1145/3406325.3451026 I-B, III-C

[32] C. L. Mallows, "Patience sorting," *Bull. Inst. Math. Appl.*, vol. 9, pp. 216–224, 1973. I

[33] M. Mitzenmacher and S. Seddighin, "Dynamic algorithms for lis and distance to monotonicity," in *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 671–684. [Online]. Available: https://doi.org/10.1145/3357713.3384240 I-B, III-C

[34] ——, "Improved sublinear time algorithm for longest increasing subsequence," in *Proceedings of the Thirty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '21. USA: Society for Industrial and Applied Mathematics, 2021, p. 1934–1947. I, III

[35] T. Naumovitz and M. Saks, "A polylogarithmic space deterministic streaming algorithm for approximating distance to monotonicity," in *Proceedings of the 2015 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2015, pp. 1252–1262. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611973730.83 I-B, I-B

[36] T. Naumovitz, M. Saks, and C. Seshadhri, *Accurate and Nearly Optimal Sublinear Approximations to Ulam Distance*, 2017, pp. 2012–2031. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611974782.131 I-B

[37] I. Newman and N. Varma, "New Sublinear Algorithms and Lower Bounds for LIS Estimation," in *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), N. Bansal, E. Merelli, and J. Worrell, Eds., vol. 198. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 100:1–100:20. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/14169 I, III

[38] N. S. Nosatzki, "Approximating the longest common subsequence problem within a sub-polynomial factor in linear time," *ArXiv*, vol. abs/2112.08454, 2021. I, I-B

[39] "List of open problems in sublinear algorithms: Problem 44," http://sublinear.info/44. I-B

[40] R. K. S. Pallavoor, S. Raskhodnikova, and E. Waingarten, *Approximating the Distance to Monotonicity of Boolean Functions*, 2020, pp. 1995–2009. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611975994.123 I

[41] M. Parnas, D. Ron, and R. Rubinfeld, "Tolerant property testing and distance approximation," *Journal of Computer and System Sciences*, vol. 72, no. 6, pp. 1012 – 1042, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000006000444 I

[42] A. Rubinstein, S. Seddighin, Z. Song, and X. Sun, "Approximation algorithms for lcs and lis with truly improved running times," in *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2019, pp. 1121–1145. I, III, III, III-A, III-C, III-C, III-D

[43] M. Saks and C. Seshadhri, "Space efficient streaming algorithms for the distance to monotonicity and asymmetric edit distance," in *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '13. USA: Society for Industrial and Applied Mathematics, 2013, p. 1698–1709. I-B

[44] M. E. Saks and C. Seshadhri, "Estimating the longest increasing sequence in polylogarithmic time," *SIAM J. Comput.*, vol. 46, no. 2, pp. 774–823, 2017. I, I-B

[45] X. Sun and D. P. Woodruff, "The communication and streaming complexity of computing the longest common and increasing subsequences," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07. USA: Society for Industrial and Applied Mathematics, 2007, p. 336–345. I