Fast Algorithms for Bounded-Range LIS Approximation

by

Pachara Sawettamalya

B.S., Mathematics and Computer Science and Engineering,
Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author . . . .
Department of Electrical Engineering and Computer Science
May 6, 2022

Certified by . . . .
Ronitt Rubinfeld
Professor
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Fast Algorithms for Bounded-Range LIS Approximation

by

Pachara Sawettamalya

Submitted to the Department of Electrical Engineering and Computer Science
on May 6, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We introduce an improvement to additive approximation of Longest Increasing Subsequence (LIS) of a sequence with a bounded number of unique elements. In particular, for a sequence f of length n with r unique elements and ϵ additive error paramenter, we present an algorithm that approximate the size of f's LIS within $\pm\epsilon n$ using $O(r\epsilon^{-2}) \cdot poly(\log \epsilon^{-1})$ samples and $O(r\epsilon^{-2}) \cdot poly(\log r, \log \epsilon^{-1})$ runtime. Our approache introduces small adjustments to the previously known algorithm for this problem, due to [5], resulting in a polynomial runtime algorithm which uses less queries by a factor of ϵ^{-1}. Similar approaches can also be applied to estimating edit distance to monotonicity in 2-dimenstional array and L_1 edit distance of a sequence within sublinear time using $poly(r, \epsilon^{-1})$ queries.

Thesis Supervisor: Ronitt Rubinfeld
Title: Professor

Acknowledgments

First and foremost, I would like to thank my parents for their constant support. Spending the majority of the past six years abroad, I have left at home two people who love me the most: mom and dad. Looking into my empty room day after day must have taken an emotional toll on them, and I cannot imagine how difficult it would be. Thank you for your patience and sacrifice, which make me who I am today. Above all, my parents always provide every care, support, and counsel I need, making sure I still feel loved every moment, no matter where I am. I owe my success to you, and I hope I make you proud today.

I would also like to extend my deepest gratitude to my research advisor Professor Ronitt Rubinfeld for agreeing to supervise my research for the past two years. Ronitt is, without a doubt, one of the most academically knowledgeable people I have ever known. But beyond that, she is also kind, caring, enduring, and understanding. Learning from you has enlightened my thoughts, and your guidance has opened up many avenues for my future. For that, I also thank you.

Finally, I consider myself fortunate enough to be surrounded by a group of good friends in the Boston area. Thank you all for making my six years in Boston a lively one, and I will forever hold fond memories of our time together.

Contents

List of Figures

Chapter 1

Introduction

The problem of finding the exact length of the longest increasing subsequence (LIS) of an arbitrary sequence f of length n, denoted by $\mathrm{LIS}(f)$, is known to be solvable in $O(n \cdot \log n)$ using dynamic programming [3]. However, the algorithm posits two major drawbacks. The first one being that computing the exact size of LIS is rather expensive – roughly in linear time which can be bad if the input is big. Moreover, if we *expect* the input sequence to be mostly sorted, it should not take as long as linear time to find $|\mathrm{LIS}(f)|$. In other words, we can assume that most elements to already be in-order, so we only need to verify the out-of-order elements which are expected to be only a small portion of the input sequence.

Second, an arbitrary sequence does not impose a constraint on the size of values pool. If the sequence consists of $\Theta(n)$ unique elements, it can take up $\Theta(n)$ time and space to represent the LIS alone. Analogously, if we limit the pool size to r, we can represent the LIS using just $O(r)$ time and space by specifying the position of those (up to) r transitions of values in the LIS.

These dilemmas lead us to the study of a sublinear time algorithms for approximating the length of LIS of a given sequence. Being able to approximate the size of LIS will potentially make an impact on closely related problems such as estimating distance to monotonicity [1], estimating length of the longest common subsequence [6], or even property testing [2]. In particular, we raise the following question:

Problem 1. Given a sequence f of n numbers from a sized-r value pool, and the target value OPT being the length of f's longest increasing subsequence (LIS). How do we propose an algorithm that outputs an approximation $\widetilde{L}$ of the target within a small error range? Ideally we want the error to be ϵ-additive – meaning that $|\widetilde{L} - \mathsf{OPT}| \leq \epsilon n$

Several attempts has been done to estimate such value. [7] gave a $(1+\epsilon)$-multiplicative approximation algorithm that runs in polylogarithmic time under the assumption that the length of LIS is at least a constant factor of the sequence's length. [6] relaxed this assumption – giving a $O(\lambda^3)$ multiplicative approximation algorithm that runs in truly sublinear time when $\lambda = |\text{LIS}(f)|/n$. Later [4] improved the approximation factor to $O(\lambda^\epsilon)$ for arbitrary $\epsilon > 0$ while maintaining sublinear runtime. In a similar fashion, we can view the problem of estimating the size of LIS as a problem of estimating the hamming distance to monotonicity. [1] gives a $(2+\epsilon)$-multiplicative approximation of such distance that runs in $O(\log n)$ with constant success probability; however, this does not guarantees the same multiplicative approximation for $|\text{LIS}(f)|$.

Perhaps one of the most recent developments are due to [5] where the authors presented an ϵ-additive approximation algorithm using $\widetilde{O}(r\epsilon^{-3})$ samples. Nevertheless, no a comprehensive runtime analysis was provided, which may be exponentially large due to the nature of their algorithm. In our paper, we introduce a small tweak to their algorithm that can bring the sample complexity down to $\widetilde{O}(r\epsilon^{-2})$ – improving by a factor of $1/\epsilon$. In particular, we make adjustment to the algorithm in a fashion that we do less samplings, but more in-depth analysis yields similar results. Moreover, we present a detailed analysis of the algorithm which guarantees not only sample complexity but also runtime of $\widetilde{O}(r\epsilon^{-2})$. The implementation relies on dynamic programming which will be covered later in section 4. Formally, we present the following theorem.

Theorem 2. Given an additive error parameter $\epsilon < 1$, confidence parameter $\delta < 1$, and a sized-r value pool $\mathcal{V}$. There exists an algorithm that takes in a sequence f of length n whose values are from $\mathcal{V}$, and output the approximation of $|\text{LIS}(f)|$ within $\epsilon \cdot n$ additive error. The algorithm succeeds with probability at least $1-\delta$, have sample complexity

$\Theta(r\epsilon^{-2}\log\epsilon^{-1}+\epsilon^{-2}\log\delta^{-1})$, and runtime $O((r\epsilon^{-2}\log\epsilon^{-1}+\epsilon^{-2}\log\delta^{-1})\,(\log r+\log\epsilon^{-1}))$.

Finally, we give several extension of the algorithms – showing similar algorithms can be used to approximate edit distance (to monotonicity) in 2-dimensional arrays and L_1 distance (to monotonicity) within small additive error using only $poly(r, \epsilon^{-1})$ samples.

Chapter 2

Notations and Preliminaries

In this section, we define the notation used in this paper. For simplicity we will assume that the pool of values of size r is $[r]$. Our length-n sequence f can be viewed as a function $f : [n] \to [r]$. For a (sub)sequence S, we let $|S|$ represent its length. For two (sub)sequences S, T, we let $S \cap T$ be their common subsequence consisting of elements which appear at the exact same positions in both S and T. With these notations, we observe that $|S \cap T|$ is the number of elements that appear at the same position in both S and T.

Given a sequence f, we call B a *block* of f if it occupies consecutive elements in f. For a block B and a value $v \in [r]$, we let $N(B, v)$ be the number of occurrences of v in B. For a (sub)sequence S and a block B of f, we let *truncate*(S, B) represent the part of S that occurs within B.

In section 3, we call $\mathcal{L} = (l_1, ..., l_t) \in [r]^t$ a *label* iff $1 \leq l_1 \leq ... \leq l_t \leq r$. Note that this notation is borrowed from [5]. Finally, suppose that we partition f into consecutive blocks $(B_1, ..., B_t) := \mathcal{B}$. We let a *pseudosolution* with respect to the sequence f, blocks $\mathcal{B}$, and label $\mathcal{L}$, denoted, *pseudosol*$(f, \mathcal{B}, \mathcal{L})$ be a subsequence of f which contains all l_1's from B_1, all l_2's from B_2, and so on. Because $l_1 \leq ... \leq l_t$, we then have *pseudosol*$(f, \mathcal{B}, \mathcal{L})$ being an increasing subsequence.

In section 5, we explore an extension of LIS approximation into a 2-dimensional array setting. In particular, suppose we have an $m \times n$ array $A \in [r]^{m \times n}$. We say that A is *increasing* iff $A(i,j) \leq A(i',j')$ for any $(i,j) \lesssim (i',j')$, meaning that $i \leq i'$ and $j \leq j'$. We call P a *partial array* of A iff P consists of some entries of A. Informally, we can think of an array as a table, and a partial array occupies some entries of such table. A partial array P is *increasing* iff $A(i,j) \leq A(i',j')$ for any $(i,j) \lesssim (i',j')$ where (i,j) and (i',j') belongs to P. Finally, given an array A, say P is A's *Largest Increasing Partial-Array*, denoted LIPA(A), iff there is no increasing partial array whose size is larger than that of P.

We note the the notations of *block*, *label*, *pseudosolution*, and *agreement* can be extended to higher dimensional arrays. Specifically, given a 2-dimensional array A, call B a *block* of A if it occupies a rectangular portion of A. Suppose that we partition A into consecutive blocks $p \times q$ blocks $\{B_{i,j} \mid (i,j) \in [p] \times [q]\} := \mathcal{B}$. Call $\mathcal{L} \in [r]^{p \times q}$ a *label* iff $\mathcal{L}_{i,j} \leq \mathcal{L}_{i',j'}$ for any $(i,j) \lesssim (i',j') \in [p] \times [q]$. Finally, we let a *pseudosolution* with respect to the array A, blocks $\mathcal{B}$, and label $\mathcal{L}$, denoted, *pseudosol*$(A, \mathcal{B}, \mathcal{L})$ be a partial array of A which contains all $\mathcal{L}_{i,j}$'s from $B_{i,j}$ for any $(i,j) \in [p] \times [q]$.

Chapter 3

Additive LIS Approximation Algorithm

The core idea of our algorithm is as follows. We first split the input sequence f into t equal consecutive blocks called $(B_1, ..., B_t) := \mathcal{B}$. For each block B_i, we will take s samples uniformly and independently called S_i, and record the number of occurrences of each number in $[r]$. Informally, the samples S_i approximately represent the distribution of values in B_i. We collectively call the set of all ts samples taken $\mathcal{S} := (S_1, ..., S_t)$.

Our goal is to, for every possible label $\mathcal{L} = (l_1, ..., l_t)$, approximate $|pseudosol(f, \mathcal{B}, \mathcal{L})|$ within a small additive error. To do so, for each $i \in [t]$, we denote $agreement(B_i, l_i, S_i)$ to be the number of occurrences of l_i among the s samples S_i taken from block B_i. We further denote denote $agreement(\mathcal{B}, \mathcal{L}, \mathcal{S}) = \sum_{i \in [t]} agreement(B_i, l_i, S_i)$ to be the number of agreements of samples $\mathcal{S}$ with pseudosolution with respect to label $\mathcal{L} = (l_1, ..., l_t)$, i.e. $pseudosol(f, \mathcal{B}, \mathcal{L})$. We then claim that, $agreement(\mathcal{B}, \mathcal{P}, \mathcal{S})$, up to a normalization factor, approximates $pseudosol(f, \mathcal{B}, \mathcal{P})$.

Below, we propose our algorithm. It is worth remarking that this algorithm does yield the desired query complexity, but not the desired runtime, but it is more understandable and easier to analyze correctness and error probability. Later on in section 4, we will improve it to the desired runtime. Our algorithm resembles that of [5], but we change parameters and structure of the algorithm to get a slightly better query complexity.

Algorithm 1 : LIS-Approximate

Input: $r \in \mathbb{Z}^+$, length-n sequence f where each entry in the sequence is an element of $[r]$, additive error parameter ϵ, confidence parameter δ

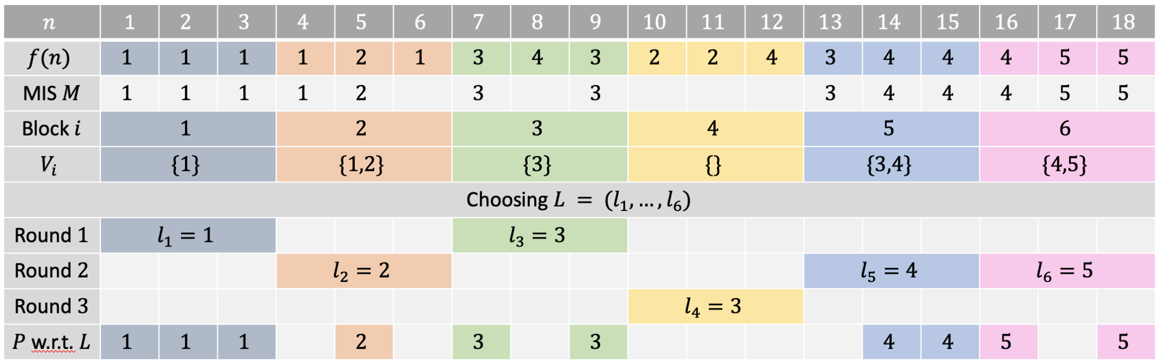
```
1: $t \leftarrow 100r/\epsilon$
2: Choose $s$ such that $ts = 30000\epsilon^{-2} \cdot (r \log(100e\epsilon^{-1}) + \log(2\delta)^{-1})$
3: $A \leftarrow$ empty $t \times r$ array
4: split $f$ into $t$ equal consecutive blocks $B_1, ..., B_t$
5: for each block $B_i$ do
6:     uniformly and independently sample $s$ elements from $B_i$
7:     for each $j \in [r]$ do
8:         $A(i,j) \leftarrow$ the number of occurrences of $j$ in those $s$ samples
9: for every possible label $\mathcal{L} = (l_1, ..., l_t)$ do
10:     calculate $agreement(\mathcal{B}, \mathcal{L}, \mathcal{S}) = \sum_{i\in[t]} A(i, l_i) = \sum_{i\in[t]} agreement(B_i, l_i, S_i)$
11: $\mathcal{L}_{max} \leftarrow$ label with largest $agreement(\mathcal{B}, \cdot, \mathcal{S})$
12: $\widetilde{L} \leftarrow \frac{n}{ts} \cdot agreement(\mathcal{B}, \mathcal{L}_{max}, \mathcal{S})$
13: output $\widetilde{L}$
```

The correctness of this algorithm is equivalent to the following lemma.

Lemma 3. Let $\mathsf{OPT} = \text{LIS}(f)$ be f's LIS with size $\mathsf{OPT}_{\mathsf{VAL}} = |\mathsf{OPT}|$. Then with probability at least $1 - \delta$, we have $\widetilde{L}$ outputted by Algorithm 1 satisfies $|\widetilde{L} - \mathsf{OPT}_{\mathsf{VAL}}| \leq \epsilon n$.

To prove this Lemma, we will instead prove a series of smaller claims. Our proof strategy is based on the fact that we only need to consider *maximal* increasing subsequences (MIS) of f because LIS(f) must be maximal. Furthermore, any non-maximal increasing subsequence can be extended by adding in-order elements to it until it is maximal. We claim that for every MIS M, there exists a pseudosolution that is close to it in size, and vice versa. Therefore, to approximate LIS, which is the longest MIS, we can find instead the largest pseudosolution.

On a high level, we will show that for every MIS M there exists some pseudosolution $\mathcal{P}_M$ for which M and $\mathcal{P}_M = pseudosol(f, \mathcal{B}, \mathcal{L}_M)$ are close in size (Claim 4). Then Claim 6



n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$f(n)$	1	1	1	1	2	1	3	4	3	2	2	4	3	4	4	4	5	5
MIS M	1	1	1	1	2		3		3				3	4	4	4	5	5
Block i		1			2			3			4			5			6	
V_i		{1}			{1,2}			{3}			{}			{3,4}			{4,5}	
Choosing $L = (l_1, \dots, l_6)$																		
Round 1		$l_1 = 1$						$l_3 = 3$										
Round 2					$l_2 = 2$									$l_5 = 4$			$l_6 = 5$	
Round 3											$l_4 = 3$							
P w.r.t. L	1	1	1		2		3		3					4	4	5		5

Figure 3-1: Examples of the construction of L and P from M. A sequence f of length 18, which is split into six blocks of three consecutive elements, is given (row 1) along with its MIS M (row 2). Each block i is associated with V_i which is the set of its unique elements appearing in M (row 3). To construct $\mathcal{L} = (l_1, ..., l_6)$, we iterate for three rounds. Round 1 assigns those l_i whose $|V_i| = 1$ (row 6). Round 2 assigns those l_i whose $|V_i| \geq 2$ (row 7). Round 3 assigns those l_i whose $|V_i| = 0$ (row 8). Finally, we can construct pseudosolution $\mathcal{P}$ from the label $\mathcal{L}$ just created (row 9).

shows that with high probability every pseudosolution $\mathcal{P} = pseudosol(f, \mathcal{B}, \mathcal{L})$ has its size approximately $\approx \frac{n}{ts} \cdot agreement(\mathcal{B}, \mathcal{L}, \mathcal{S})$. Therefore, we should be able to approximate

$$\begin{aligned} \mathsf{OPT}_{\mathsf{VAL}} = |\mathrm{LIS}(f)| = \max_{\text{MIS } M} |M| &\approx \max_{\text{label } \mathcal{L}} |pseudosol(f, \mathcal{B}, \mathcal{L})| \\ &\approx \frac{n}{ts} \cdot agreement(\mathcal{B}, \mathcal{L}_{max}, \mathcal{S}) \\ &= \widetilde{L}. \end{aligned}$$

Claim 8 offers an analysis of the magnitude of additive error $|\widetilde{L} - \mathsf{OPT}|$.

Claim 4. For each of f's maximal increasing subsequences M, there *exists* a pseudosolution $\mathcal{P}$ defined by a labeling $\mathcal{L} = (l_1, ..., l_t)$, i.e. $\mathcal{P} = pseudosol(f, \mathcal{B}, \mathcal{L})$, such that $||M| - |\mathcal{P}|| \leq rn/t$, where r, n, and t are given in algorithm 1. With the parameters specified in Algorithm 1, the bound becomes $\epsilon n/100$.

Proof. Recall that we split f into t blocks $(B_1, ..., B_t) := \mathcal{B}$. For each $i \in [t]$, let V_i be the set of its values that appear in $M \cap B_i$.

We will then build $\mathcal{L} = (l_1, ..., l_t)$ by iterating over $V_1, ..., V_t$ for 3 rounds. In each round we do as follows (see Figure 3-1.)

- Round 1: for those V_i with $|V_i| = 1$, set l_i to be the (only) element of V_i.
- Round 2: for those V_i with $|V_i| \geq 2$, set l_i to be an arbitrary element of V_i
- Round 3: for those V_i with $|V_i| = 0$, set l_i to be the closest l_j that has already been set.

We first need to verify that $\mathcal{L} = (l_1, ..., l_t)$ is a valid labeling – i.e. $l_1 \leq ... \leq l_t$. Notice that in round 1 and round 2, we assign values to l_i from V_i. As V_i's are in non-decreasing order (in accordance with M), there is no out-of-order l_i's amongst those $|V_i| \neq 0$. In round 3, we fill in those l_i's for $|V_i| \neq 0$ with their closest already-been-set l_j, we do not cause any order violation. Therefore, the ordering $l_1 \leq ... \leq l_t$ is guaranteed.

For those $|V_i| = 1$, we notice that M and $\mathcal{P}$ are identical within B_i, as the label of $\mathcal{P}$ for block B_i is the only element in $M \cap B_i$.

For those $|V_i| = 0$, it means that M does to take any elements in B_i. If $l_i = l_j$ did appear in B_i, we could have extended M by adding in those l_j's in B_i which contradicts to the maximality of M. Therefore, within block B_i, no l_i appears as well as M, meaning that M and $\mathcal{P}$ are identical within B_i,

For those $|V_i| \geq 2$, within B_i, we have M and $\mathcal{P}$ differed by at most n/t elements since they both are contained in a sized-(n/t) block B_i.

We also notice that there are at most $r-1$ blocks that make $|V_i| \geq 2$ since two consecutive blocks of this type B_i and B_j can share at most 1 element in V_i and V_j (tail of prior block and head of latter block) and the union of these V's have to have at most r elements. An alternative-but-informal way to view this argument is to consider the worst case scenario when $V_i = \{i, i+1\}$ for $i \in [r-1]$. So we can have at most $r-1$ blocks with $|V_i| \geq 2$.

Therefore, we have $||M| - |\mathcal{P}|| \leq (r-1) \cdot \frac{n}{t} \leq \epsilon n/100$, as wished.

$\square$

Definition 5. Say a pseudosolution $\mathcal{P}$ of label $\mathcal{L}$ is *bad* (with respect to samples $\mathcal{S}$) iff $\left||\mathcal{P}| - \frac{n}{ts} \cdot \mathit{agreement}(\mathcal{B}, \mathcal{L}, \mathcal{S})\right| > \epsilon n/100$. In other words, the agreement of $\mathcal{L}$ with the samples $\mathcal{S}$ does not approximate $\mathcal{P}$.

The following claim argues that any pseudosolution is bad with small probability.

Claim 6. For any pseudosolution $\mathcal{P}$, we have $\Pr(\mathcal{P} \text{ is bad}) \leq 2 \cdot \exp(-\epsilon^2 ts/30000)$.

Proof. Let $\mathcal{P}$ be a pseudosolution with respect to label $\mathcal{L} = (l_1, ..., l_t)$.

For any $i \in [t]$ and $j \in [s]$, denote $Z_{i,j}$ to be an indicator random variable of the following event: the j^{th} sample of S_i is l_i. This means $A(i, l_i) = \sum_{j \in [s]} Zi, j$. Moreover, all the $Z_{i,j}$'s are independent.

We realize that

$$\mathit{agreement}(\mathcal{B}, \mathcal{L}, \mathcal{S}) = \sum_{i \in [t]} A(i, l_i) = \sum_{i \in [t]} \sum_{j \in [s]} Z_{i,j}.$$

This means we can view $\mathit{agreement}(\mathcal{B}, \mathcal{P}, \mathcal{S})$ as a sum of ts independent Bernoulli random variables although their biases are varied. We then can use Chernoff bound to upper bound the probability that $\mathit{agreement}(\mathcal{B}, \mathcal{P}, \mathcal{S})$ deviates much from its mean. We also realize that

$$\begin{aligned}
\mathbb{E}\left[\mathit{agreement}(\mathcal{B}, \mathcal{L}, \mathcal{S})\right] &= \sum_{i \in [t]} \sum_{j \in [s]} \mathbb{E}(Z_{i,j}) \\
&= \sum_{i \in [t]} \sum_{j \in [s]} \frac{|\mathcal{P} \cap B_i|}{|B_i|} \\
&= \frac{ts}{n} \cdot \sum_{i \in [t]} |\mathcal{P} \cap B_i| \\
&= \frac{ts}{n} \cdot |\mathcal{P}|.
\end{aligned}$$

We note that $|\mathcal{P}| \leq n$ as $\mathcal{P}$ is a subsequence of f which has length at most n. Therefore, we can evaluate:

$$\begin{aligned}
\Pr(\mathcal{P} \text{ is bad}) &= \Pr\left(||\mathcal{P}| - \frac{n}{ts} \cdot agreement(\mathcal{B}, \mathcal{L}, \mathcal{S})| > \epsilon n/100\right) \\
&= \Pr\left(\left|\frac{ts}{n} \cdot |\mathcal{P}| - agreement(\mathcal{B}, \mathcal{L}, \mathcal{S})\right| > \frac{\epsilon ts}{100}\right) \\
&= \Pr\left(|agreement(\mathcal{B}, \mathcal{L}, \mathcal{S}) - \mathbb{E}(agreement(\mathcal{B}, \mathcal{L}, \mathcal{S}))| > \frac{\epsilon ts}{100}\right) \\
&\leq 2 \cdot \exp\left(-\frac{\epsilon^2 t^2 s^2}{30000 \cdot \mathbb{E}[agreement(\mathcal{B}, \mathcal{L}, \mathcal{S})]}\right) \\
&= 2 \cdot \exp\left(-\frac{\epsilon^2 t^2 s^2}{30000 \cdot (ts/n) \cdot |\mathcal{P}|}\right) \\
&\leq 2 \cdot \exp\left(-\frac{\epsilon^2 t^2 s^2}{30000 \cdot (ts/n) \cdot n}\right) \\
&= 2 \cdot \exp\left(-\frac{\epsilon^2 ts}{30000}\right)
\end{aligned}$$

, as wished. □

Claim 7. With parameters specified in Algorithm 1, the probability that there's no bad pseudosolutions is at least $1 - \delta$.

Proof. We first ask how many pseudosolutions, i.e. labels, there are. We realize that in order for $\mathcal{L} = (l_1, ..., l_t)$ to be a pseudosolution, it is sufficient and necessary that $l_1 \leq ... \leq l_t$ are chosen from $\{1, ..., r\}$. Choosing $(l_1, ..., l_t)$ is equivalent to choosing to put $r - 1$ bars into t slots $B_1, ..., B_t$ so that $r - 1$ bars split consecutive blocks $B_1, ..., B_t$ into r groups (can be empty) which group i is assigned value i. Thus, the number of pseudosolutions is upper-bounded by

$$\binom{t+r-1}{r-1} \leq \left(\frac{e(t+r-1)}{r-1}\right)^{r-1} \approx (100e\epsilon^{-1})^{r-1} \approx \exp(r \cdot \log(100e\epsilon^{-1}))$$

Thus, from the union bound, the probability that there's no such bad pseudosolution is at most

$$\begin{aligned}\Pr(\text{bad pseudosolution exists}) &\leq 2 \cdot \exp\left(-\frac{\epsilon^2 ts}{30000}\right) \cdot \exp(r \cdot \log(100e\epsilon^{-1})) \\ &= 2 \cdot \exp\left(r \cdot \log(100e\epsilon^{-1}) - \frac{\epsilon^2 ts}{30000}\right)\end{aligned}$$

By choosing $ts = 30000\epsilon^{-2} \cdot (r \log(100e\epsilon^{-1}) + \log(2\delta)^{-1}) = O(r\epsilon^{-2} \log \epsilon^{-1} + \epsilon^{-2} \log \delta^{-1})$, such probability is upper bounded by δ as desired. $\square$

Claim 8. If there are no bad pseudosolutions, then $\left|\widetilde{L} - \mathsf{OPT}_{\mathsf{VAL}}\right| \leq \epsilon n$.

Proof. Let $\mathcal{P}_{max} = pseudosol(f, \mathcal{B}, \mathcal{L}_{max})$ be the pseudosolution with respect to $\mathcal{L}_{max}$. First we recall that $\mathcal{P}$ not being *bad* means $||\mathcal{P}_{max}| - \frac{n}{ts} \cdot agreement(\mathcal{B}, \mathcal{L}_{max}, \mathcal{S})| \leq \epsilon n/100$. This implies $\widetilde{L} = \frac{n}{ts} \cdot agreement(\mathcal{B}, \mathcal{L}_{max}, \mathcal{S}) \leq |\mathcal{P}_{max}| + \epsilon n/100 \leq \mathsf{OPT}_{\mathsf{VAL}} + \epsilon n/100$ since $|\mathcal{P}_{max}|$ is an increasing subsequence in f whose length has to be at most $\mathsf{OPT}_{\mathsf{VAL}} = |\mathrm{LIS}(f)|$.

Denote $\mathcal{P}_{\mathsf{OPT}}$ to be a pseudosolution that $||\mathsf{OPT}| - |\mathcal{P}_{\mathsf{OPT}}|| \leq \epsilon n/100$. Since OPT is an LIS which in turn is maximal, the existence $\mathcal{P}_{\mathsf{OPT}}$ is given by Claim 4. Let $\mathcal{L}_{\mathsf{OPT}}$ be the label corresponding to $\mathcal{P}_{\mathsf{OPT}}$. The assumption that $\mathcal{P}_{\mathsf{OPT}}$ is not bad tells us that $||\mathcal{P}_{\mathsf{OPT}}| - \frac{n}{ts} \cdot agreement(\mathcal{B}, \mathcal{L}_{\mathsf{OPT}}, \mathcal{S})| \leq \epsilon n/100$. From triangle inequality, we then have $||\mathsf{OPT}| - \frac{n}{ts} \cdot agreement(\mathcal{B}, \mathcal{L}_{\mathsf{OPT}}, \mathcal{S})| \leq \epsilon n/50$. We then have $\frac{n}{ts} \cdot agreement(\mathcal{B}, \mathcal{L}_{\mathsf{OPT}}, \mathcal{S}) \geq |\mathsf{OPT}| - \epsilon n/50 = \mathsf{OPT}_{\mathsf{VAL}} - \epsilon n/50$. Recall that $\mathcal{L}_{max}$ is the label with largest $agreement(\mathcal{B}, \cdot, \mathcal{S})$. Therefore, $\widetilde{L} = \frac{n}{ts} \cdot agreement(\mathcal{B}, \mathcal{L}_{max}, \mathcal{S}) \geq \frac{n}{ts} \cdot agreement(\mathcal{B}, \mathcal{L}_{\mathsf{OPT}}, \mathcal{S}) \geq \mathsf{OPT}_{\mathsf{VAL}} - \epsilon n/50$.

Therefore we now have $-\epsilon n/100 \leq \widetilde{L} - \mathsf{OPT}_{\mathsf{VAL}} \leq \epsilon n/50$, which means $|\widetilde{L} - \mathsf{OPT}_{\mathsf{VAL}}| \leq \epsilon n/50 \leq \epsilon n$ as wished. $\square$

Combining Claim 8 and Claim 7 proves Lemma 3 – as there's probability at least $1 - \delta$ that no bad pseudosolution exists, which implies $\left|\widetilde{L} - \mathsf{OPT}_{\mathsf{VAL}}\right| \leq \epsilon n$.

Chapter 4

Improving Runtime via Dynamic Programming

In this section, we address two major issues of Algorithm 1 that prevent us from getting a fast runtime. The first issue is the exhaustive search of $\mathcal{L}_{max}$ which a naive implementation requires iterations over an exponential number of labels. The second issue is the overkilling process of filling up T. We make a crucial reservation that A is sparse, meaning that a large fraction of A is 0. Moreover, those zero entries of A has little to no no use in updating entries of T. Thus, it should suffice to store only non-zero entries of A and update T properly.

The first issue is resolved by using dynamic programming (section 4.1) and the second issue is resolved by restructuring A so that it only records non-zero entries and implementing a special data structure to store T (section 4.2).

4.1 Dynamic Programming Framework

In Algorithm 1, we determine $\mathcal{L}_{max}$ by enumerating over all possible pseudosolutions. However, the total number of pseudosolution is, due to Claim **??**, approximately $\exp(r \cdot \log(100e\epsilon^{-1}))$ which is highly undesirable. Therefore we need to circumvent the exhaustive search. We will resolve this problem by using dynamic programming. To be precise, we restate the problem of finding $\mathcal{L}_{max}$ as follows. Each block B_i has r non-negative values $A(i,1), ..., A(i,r)$ for which $A(i,j)$ records $agreement(B_i, S_i, j)$ = the number of occurrences of j among the s samples S_i. We then want to determine the non-decreasing sequence (label) $\mathcal{L}_{max} = (l_1, ..., l_t) \in [r]^t$ that maximizes $\sum_{i \in t} A(i, l_i)$. We propose the following algorithm.

The algorithm uses dynamic programming for finding $\mathcal{L}_{max}$. It assigns $l_1 \leq l_2 \leq ... \leq l_t$ in order, and constructs a table T which records $T(i,v)$ which is, under a constraint that $l_i = v$, the maximum summation of $\sum_{1 \leq j \leq i} A(j, l_j)$. In other words, $T(i,v)$ is the length of longest increasing subsequence among the first i blocks conditioned that $l_i = v$. Once the table T is full, we can recover $agreement(\mathcal{L}_{max})$ by looking up the maximum value in the array T, and can reconstruct $\mathcal{L}_{max}$ by backtracking assignments at each of t blocks. The correctness of Algorithm 2 is justified with its base cases and updating rules as follows.

Algorithm 2 : Find-Lmax-Dense(A)

```
1: $T \leftarrow$ empty $t \times r$ array initialized with 0's
2: for $v \in [r]$ do
3:     $T(1,v) \leftarrow A(1,v)$
4: initialization of the first block
5: for $i = 1, ..., t-1$ do
6:     for $v = 1, ..., r$ do
7:         $T(i+1,v) \leftarrow A(i+1,v) + \max_{1 \leq u \leq v} T(i,u)$
8: $(\tau, \gamma) \leftarrow \max_{\tau \in [t], \gamma \in [r]} T(\tau, \gamma)$
9: output the label $\mathcal{L}_{max} = (l_1, ..., l_t)$ that yields $T(\tau, \gamma)$
```

Theorem 9. Algorithm 2 correctly determines $\mathcal{L}_{max}$ and runs in $O(rt)$ time.

Proof. Notice that we begin with T empty, and will constantly fill up the array $T(i,v)$ in in increasing order of (i,v). Moreover, once we assign a value to $T(i,v)$, it remains unchanged throughout the algorithm. We will prove by strong induction the following statement, denoted by $H(i, v)$.

$$T(i, v) = \max_{\substack{(l_1,\ldots,l_i=v)\in[r]^i \\ l_1\leq\ldots\leq l_i=v}} \sum_{j\in[i]} A(j, l_j).$$

For base cases, notice that $H(1, v)$ means $T(i, v) = A(1, v)$ which is trivially true due to lines 2-3 of the algorithm.

Now we need to justify the induction step. Assume that $H(i', w)$ is true for all $i' \leq i$ and $w \in [r]$. For any $v \in [r]$, we can determine $T(i + 1, v)$ as follows.

$$\begin{aligned}
T(i+1, v) &= A(i+1, v) + \max_{1\leq u\leq v} T(i, u) \\
&= A(i+1, v) + \max_{1\leq u\leq v} \max_{\substack{(l_1,\ldots,l_i=u)\in[r]^i \\ l_1\leq\ldots\leq l_i=u}} \sum_{j\in[i]} A(j, l_j) \\
&= \max_{\substack{(l_1,\ldots,l_{i+1}=v)\in[r]^{i+1} \\ l_1\leq\ldots\leq l_{i+1}=v}} \sum_{j\in[i+1]} A(j, l_j)
\end{aligned}$$

which implies that $H(i + 1, v)$ is true. This concludes the induction step.

Now we analyze the runtime. We notice that for each i in line 4, we need to compute $\max_{1\leq u\leq v} T(i, u)$ for each $v \in [r]$. This can be done within $O(r)$ by iterating over $T(i, u)$'s an increasing order of u, and record the maximum up to that moment. Overall, each round of line 5-6 takes $O(r)$. This means the algorithm takes overall time $O(tr)$, as wished.

Finally, to derive the largest summation $\sum_{i\in t} A(i, l_i)$ over all t blocks, we can simply look at $T(t, r)$. Note that we can determine the $(l_1, ..., l_t)$ that yields $T(t, r)$ by keeping track of u for each $T(i + 1, v)$ in line 6 of the algorithm, and then backtracking from $T(t, r)$. The details of the bookkeeping is omitted for reasons of succinctness.

In total, the algorithm runs in $\Theta(tr) = \Theta(r^2\epsilon^{-1})$ – which now improves from exponential (in exhaustive search) to polynomial time. □

4.2 Restructuring A

In the previous section, we notice that the runtime is also lower-bounded by the size of A which is $t \cdot r$. However, we make another crucial observation that A is indeed sparse, meaning that only a small fraction is non-zero. This is because for any $i \in [t]$, there are at most $s = O(\epsilon^{-1})$ out of r entries of $A(i, \cdot)$ that are non-zero since we only do s samples per block. In particular, we denote $U_i \subseteq [r]$ to be the set of unique values appearing in any of the samples S_i of block B_i. In other words, those numbers in U_i occurs at least once in S_i, and those not in U_i do not occur in S_i; thus can be ignored. With this prior knowledge of A's sparsity, we can cut down the runtime of Algorithm 2 by proposing an alternative algorithm for finding $\mathcal{L}_{max}$.

Algorithm 3 : Find-Lmax-Sparse(A)

```
1: $T \leftarrow$ an empty look-up table
2: for $i \in [t]$ in increasing order do
3:     sort $U_i$ in an increasing order
4: for $v \in U_1$ in increasing order do
5:     $T(1, v) \leftarrow A(1, v)$
6: for $i = 1, ..., t-1$ do
7:     for $v \in U_{i+1}$ do
8:         $T(i+1, v) \leftarrow A(i+1, v) + \max_{\substack{j \leq i \\ u \leq v, u \in U_j}} T(j, u)$
9: $(\tau, \gamma) \leftarrow \max_{\tau \in [t], \gamma \in U_\tau} T(\tau, \gamma)$
10: output the label $\mathcal{L}_{max} = (l_1, ..., l_t)$ that yields $T(\tau, \gamma)$
```

We make a crucial note that runtime of the above algorithm is majorly dominated by operations performed on T (retrieve, update values) which we will discuss later. In this section, we will only show the correctness of the algorithm. Then in section 4.3, we will discuss the data structure which will be used to store T.

In Algorithm 2, the 0s entries of A do not contribute to maximization updates – thus can be ignored. In other words, we only care about non-zero entries of A. Therefore, we can redesign the structure of A so that it only stores non-zero entries, which are at most s entries per block (as we took s samples per block.)

On a high level, Algorithm 2 and Algorithm 3 behave in the same way with slight modifications on how to store information (A and T). But with these modifications, we no longer have to update all $T(i, v)$ for every $i \in [t]$ and $v \in [r]$. We instead only need to update $T(i, v)$ for those $v \in U_i$ which is at most s times for each i. We now justify the algorithm's correctness by discussing its updating rules.

Theorem 10. Algorithm 3 correctly determines $\mathcal{L}_{max}$.

Proof. The correctness proof of this algorithm is almost identical to that of Algorithm 2. We are going to use induction on the following hypothesis, denoted $H(i, v)$ whenever $v \in S(i)$.

$$T(i,v) = \max_{\substack{(l_1,\ldots,l_i=v)\in[r]^i \\ l_1\le\ldots\le l_i=v \\ l_1\in U_1,\ldots,l_i\in U_i}} \sum_{k\in[i]} A(k,l_k).$$

For base cases $i = 1$ and $v \in S_1$, notice that $H(1, v)$ means $T(i, v) = A(1, v)$ which is trivial due to lines 4-5. Now we need to justify the induction step. Let us assume that $H(i', w)$ is true for all $i' \le i$ and \$$w \in U_i$. For any $v \in U_{i+1}$, we can determine $T(i+1, v)$ as follows.

$$\begin{aligned}
T(i+1,v) &= A(i+1,v) + \max_{\substack{j\le i \\ u\le v, u\in U_j}} T(j,u) \\
&= A(i+1,v) + \max_{\substack{j\le i \\ u\le v, u\in U_j}} \max_{\substack{(l_1,\ldots,l_j=u)\in[r]^j \\ l_1\le\ldots\le l_j=u \\ l_1\in U_1,\ldots,l_j\in U_j}} \sum_{k\in[i]} A(k,l_k) \\
&= \max_{\substack{(l_1,\ldots,l_{i+1}=v)\in[r]^{i+1} \\ l_1\le\ldots\le l_{i+1}=v \\ l_1\in U_1,\ldots,l_{i+1}\in U_{i+1}}} \sum_{k\in[i+1]} A(k,l_k)
\end{aligned}$$

which implies that $H(i+1, v)$ is true. This concludes the induction step. □

We can use the same bookkeeping technique to retrieve $\mathcal{L}_{max}$ once all $T(i, v)$'s are determined. The runtime bottleneck of our algorithm is the derivation of $\max_{\substack{j \leq i \\ u \leq v, u \in U_j}} T(j, u)$ for every $u \in U_j$. Naive implementation requires runtime as high as $O(t^2 s)$. To reduce the runtime, we first propose a useful data structure in which we call Maximum-Ordered Oracle.

4.3 Maximum-Ordered Oracle

In this section, we will discuss a special kind of oracle/data structure that will be useful in reducing the runtime of Algorithm 3. In particular, we will use the oracle for storing T which allows quick retrieval and updates.

Suppose that we have r bins. Each bin contains one number at a time which can be updated. Initially, at time $t = 0$, each bin i starts with a number $\beta_i := M_{i,0}$. At each timestep, we are allowed to *increase* the number in exactly one of the bins.

Formally, we denote $M_{i,t}$ to be the number in bin i at timestep t (begins with $t = 0$). At timestep t, we can change the number in an arbitrary bin i to c for any $c > M_{i,t}$. As we only change exactly one bin for each timestep, we will have $M_{i,t+1} = c$ and $M_{j,t+1} = M_{j,t}$ for any $j \neq i$.

Define *cumulative maximums* of time t to be an r-tuple $(m_i, ..., m_r)$ where $m_i = \max_{j \leq i} M_{j,t}$. In other words, the cumulative maximum m_i is the largest number among the first i bins. We say that the cumulative maximum *increases* at i when $m_i > m_{i-1}$, meaning that the number in bin i is strictly greater than all $i - 1$ prior bins.

We will build an oracle $\mathcal{M}$ that supports the following operations (see Figure 4-1 for demonstration.)

1. *initialize*$(\beta_1, ..., \beta_r)$
   - Initialize $\mathcal{M}$.
   - Run in $O(r)$ time.
   - Only called once at the beginning.

2. *cmax*(i)

   - Find the cumulative maximum $m_i = \max(M_{1,t}, ..., M_{i,t})$ at the current time step t.
   - Run in $O(\log r)$ time.
   - Can be called at any point.

3. *update_val*(i, v)

   - Update the value of bin i to v which is strictly greater than the current value $M_{i,t}$.
   - The execution of n consecutive *updates_val*'s takes time $O(n \log r + r)$.
   - Calling *update_val* increments a timestep. In other words, the timestep t means that there has been t value updates already performed.

In relation to Algorithm 3 for finding $\mathcal{L}_{max}$, we will have r bins where bin v contains the maximum value of $T(\cdot, v)$ that has already been assigned. For each $i = 1, ..., t-1$ in line 6, it suffices to recover the cumulative maximum only once, and use it for every $v \in U_{i+1}$. This is as opposed to multiple look-ups for each $v \in U_{i+1}$; thus should lower runtime as we desire.

Data Structure/Sketch. At any timestep k when k = number of values updates already performed, we want to keep track of all positions of changes in the cumulative maximums, along with the values they change into. That is; at time k, we record t_k tuples $(a_1, b_1), (a_2, b_2), ..., (a_{t_k}, b_{t_k})$, collectively called $\mathcal{T}_k$, for which a_i's are the positions where cumulative maximums change, and b_i is the value they changes into. In other words, for any $a_i \leq l < a_{i+1}$, we have $\max(M_{1,t}, ..., M_{l,t}) = b_i$. We also keep a_i's in an increasing order – that is $1 \leq a_1 < ... < a_{t_k} \leq r$.

Initialization. We can quickly go through $\beta_1, ...\beta_r$ and initialize the tuples $\mathcal{T}_0$ according to data structure specified above. This takes time $O(r)$.

i	1	2	3	4	5	6	7	8	9	10	T
β_i Initial value in bin i	5	2	4	6	9	9	5	12	1	15	(1,5), (4,6), (5,9), (8,12), (10,15)
$update\_val(9,10)$	5	2	4	6	9	9	5	12	10	15	(1,5), (4,6), (5,9), (8,12), (10,15)
$update\_val(4,8)$	5	2	4	8	9	9	5	12	10	15	(1,5), (4,6), (5,9), (8,12), (10,15)
$update\_val(6,13)$	5	2	4	8	9	13	5	12	10	15	(1,5), (4,6), (5,9), (6,13), (10,15)
$cmax(2)$	output 5 since $1 \leq 2 < 4$										
$cmax(5)$	output 9 since $5 \leq 5 < 6$										
$update\_val(3,14)$	5	2	14	8	9	13	5	12	10	15	(1,5), (3,14), (10,15)
$cmax(2)$	output 5 since $1 \leq 2 < 3$										
$cmax(5)$	output 14 since $3 \leq 5 < 10$										

Figure 4-1: Shown above is a demonstration of Maximum-Ordered Oracle $\mathcal{M}$ with 10 bins. Initial values $\beta_1, ..., \beta_{10}$ are given on the first row. The data structure/sketch is $\mathcal{T}$ given in the last column. Four $update_val$'s are demonstrated along with changes in $\mathcal{T}$ (value updates indicated in green entries). Finally, two $cmax$ queries are performed at each timestep $t = 3, 4$.

cmax. Look at the current $\mathcal{T}_k$. Find i that $a_i \leq v < a_{i+1}$ and output b_i. Finding such i can be done via a binary search which takes time $O(\log r)$.

Update Values. Suppose that we are at a timestep k, that is we have done k value updates so far. Look at the current $\mathcal{T}_k$. First, find j that $a_j \leq i < a_{j+1}$. This takes time $O(\log r)$ via binary search.

1. If $v \leq b_j$, do nothing.

2. If $v > b_j$, do as follows.

   2.1. Add a tuple (i, v) right after (a_j, b_j).

   2.2. For any $j' = j + 1$ to t_k, as long as $b_{j'} \leq v$, delete the tuple $(a_{j'}, b_{j'})$. Once we encounter the first occurrence of $b_{j'} > v$, we halt.

Suppose that we delete a total of d_k tuples. The runtime of deletion is thus $O(1 + d_k)$. In total, at time step k, we spend time $O(\log r + d_k)$.

Performing n *update_vals* then takes time $O(n \log r + D)$ where $D = d_0 + ... + d_{n-1}$. Now let us upper-bound D. We begin with $t_0 \leq r$ tuples at $\mathcal{T}_0$. During the n updates, we add at most n tuples (at most 1 per update) and delete D tuples. In the end, we have $t_n \geq 1$ tuples at $\mathcal{T}_n$. Therefore, we can write $1 \leq t_n \leq t_0 + n - D \leq r + n - D$ implying $D \leq r + n - 1$. This concludes that the time spent for n value updates is $O(n \log r + r)$.

4.4 Finalizing Algorithm 1

Now we can finally integrate dynamic programming into LIS-Approximation via the use of Maximum-Ordered Oracle. We first show that with the use of such oracle, Algortihm 3 (Find-Lmax-Sparse) can find $\mathcal{L}_{max}$ quickly.

Theorem 11. Using Maximum-Ordered Oracle, Algorithm 3 runs in $O(r + ts \log rs)$.

Proof. Sorting U_i for every $i \in [t]$ takes $O(ts \log s)$. The runtime bottleneck of our algorithm is the derivation of $\max_{\substack{j \leq i \\ u \leq v, u \in U_j}} T(j, u)$ for every $u \in U_j$. To do so, we will use the maximum-ordered oracle.

In particular, suppose that we have r bins so that bin v keeps track on the maximum value of $T(\cdot, v)$ that has already been assigned (this is exactly $M_{v,t}$).

The initialization is equivalent to $\beta_v = A(1, v)$ if $v \in U_1$, and 0 otherwise. The process takes $O(r)$ time.

Since we assign $T(i, v)$'s in an increasing order of i and $v \in U_i$, anytime we try to compute $\max_{\substack{j \leq i \\ u \leq v, u \in U_j}} T(j, u)$, we can simply ask for *cmax*(v). Once we have assigned $T(i + 1, v)$, we perform *update_val*$(v, T(i + 1, v))$ iff $T(i + 1, v) >$ the value currently in bin v.

In total, we perform the initialization, ts rounds of *cmax*'s, and ts rounds of *update_val*'s which takes total time $O(r + ts \log r)$. Combining with the sorting time $O(ts \log s)$, our Algorithm 3 runs in $O(r + ts \log rs)$ □

The runtime of Algorithm 3 also dominates that of Algorithm 1, meaning that Algorithm 1 runs in $O((r\epsilon^{-2} \log \epsilon^{-1} + \epsilon^{-2} \log \delta^{-1}) (\log r + \log \epsilon^{-1}))$. The number of samples we take is still $ts = \Theta(r\epsilon^{-2} \log \epsilon^{-1} + \epsilon^{-2} \log \delta^{-1})$. These, togehter, proves Theorem 2.

Chapter 5

Estimating Edit Distances in 2-Dimensional Array

We consider the *edit distance to monotonicity* $d_{\text{edit}}^{\text{mono}}(\cdot)$ of an array A (of any dimension) to be the smallest number of entries of A needing alteration in order to make A monotone. When A is a 1-dimensional array of length n, i.e. a sequence, we will have:

$$d_{\text{edit}}^{\text{mono}}(A) = \min_{\substack{\text{monotone sequence } B \\ \text{of length } n}} \left(\sum_{i \in [n]} 1_{A(i) \neq B(i)} \right)$$

On the other hand, we have $d_{\text{edit}}^{\text{mono}}(A) = |A| - |\text{LIS}(A)|$ since it suffices to change only the entries that are not in $\text{LIS}(A)$ in order to make A monotone. Therefore, to approximate edit distance in 1-dimensional array, we can use Algorithm 3 to estimate $|\text{LIS}(A)|$ and then subtract it from $|A| = n$. The approximated edit distance also has ϵn additive error.

Approximating edit distance $d_{\text{edit}}^{\text{mono}}(\cdot)$ of a 2-dimensional array can be done in a similar fashion via extensions of Theorem 2 and Algorithm 3.. In particular, given an $m \times n$ array A, we denote

$$d_{\text{edit}}^{\text{mono}}(A) = \min_{\substack{\text{monotone array } B \\ \text{of dimension } m \times n}} \left(\sum_{i \in [m], j \in [n]} 1_{A(i,j) \neq B(i,j)} \right).$$

Similar to the 1-dimensional cases, it suffices to additively approximate the size of A's

Largest Increasing Partial-Array, denoted LIPA(A), and subtract it from mn. Formally, we propose the following theorem.

Theorem 12. Let A is a $m \times n$ 2-dimensional array ($m \leq n$) whose entries are in $[r]$. Let ϵ be an additive error parameter. Then there is an algorithm that estimates $d_{\text{edit}}^{\text{mono}}(A)$ within ϵmn additive error with probability at least $1 - \delta$ using $\Theta\left(r^2\epsilon^{-3}\log\left(1 + \frac{\epsilon m}{r}\right) + \epsilon^{-2}\log\delta^{-1}\right)$ samples.

Algorithm 4 : LIS-Approximate
Input: $r \in \mathbb{Z}^+$, $m \times n$ array $f \in [r]^{m\times n}$, additive error parameter ϵ, confidence parameter δ

```
 1: $t \leftarrow 100r/\epsilon$
 2: Choose $s$ such that $tms = 30000\epsilon^{-2} \cdot (100r^2\epsilon^{-1}(1 + \log(1 + \epsilon m/100r)) + \log(2/\delta)) \approx \Theta\left(r^2\epsilon^{-3}\log\left(1 + \frac{\epsilon m}{r}\right) + \epsilon^{-2}\log\delta^{-1}\right)$
 3: split $f$ into $m \times t$ subarrays; each of size $1 \times (n/t)$; namely $B_{i,j}$ for each $i \in [m], j \in [t]$
 4: for each $B_{i,j}$ do
 5:     uniformly and independently sample $s$ elements from $B_{i,j}$
 6:     for each $k \in [r]$ do
 7:         $A(i,j,k) \leftarrow$ the number of occurrences of $k$ in those $s$ samples
 8: for every possible (monotone) label $\mathcal{L} \in [r]^{m\times[t]}$ do
 9:     calculate $agreement(\mathcal{B}, \mathcal{L}, \mathcal{S}) = \sum_{i\in[m]}\sum_{j\in[t]} agreement(B_{i,j}, l_{i,j}, S_{i,j})$
10: $\mathcal{L}_{max} \leftarrow$ label with largest $agreement(\mathcal{B}, \cdot, \mathcal{S})$
11: $\widetilde{L} \leftarrow agreement(\mathcal{B}, \mathcal{L}_{max}, \mathcal{S})$
12: output $\widetilde{L}$
```

We make a crucial note that Algorithm 4 uses $tms = \Theta\left(r^2\epsilon^{-3}\log\left(1 + \frac{\epsilon m}{r}\right) + \epsilon^{-2}\log\delta^{-1}\right)$ samples – matching the bound stated in theorem 12.

The correctness of Algorithm 4 is equivalent to the following lemma.

Lemma 13. Let OPT = LIPA(f) be f's LIPA with size $\mathsf{OPT}_{\mathsf{VAL}} = |\mathsf{OPT}|$. Then with probability at least $1-\delta$, we have $\widetilde{L}$ outputted by Algorithm 4 satisfies $|\widetilde{L} - \mathsf{OPT}_{\mathsf{VAL}}| \leq \epsilon mn$.

To justify its correctness, we will mimic the proof of correctness of Algorithm 1. We first show every *maximal* increasing partial-array (MIPA) of f has a pseudosolution that is close to it in size. Therefore, to approximate LIPA, which is the largest MIPA, we can find instead the largest pseudosolution.

For each of f's maximal increasing subsequences M, there *exists* a pseudosolution $\mathcal{P}$ defined by a labeling $\mathcal{L} = (l_1, ..., l_t)$, i.e. $\mathcal{P} = \mathit{pseudosol}(f, \mathcal{B}, \mathcal{L})$, such that $||M| - |\mathcal{P}|| \leq rmn/t$, where r, m, n, and t are given in Algorithm 4. With the parameters specified in Algorithm 1, the bound becomes $\epsilon mn/100$.

Lemma 14. Let C be a maximal increasing partial-array of A. Consider the intersection of C with each of the $m \times t$ subarrays as is step 3 in Algorithm 4. Let $d(C)$ be the number of C's subarrays whose entries contain at least two distinct numbers. Then, $d(C) \leq m(r-1)$.

Proof. Suppose, to the contrary, that $d(C) \geq m(r-1)$. This means there must exist $i \in [m]$ for which at least r subarrays among $\{B_{i,1}, ..., B_{i,t}\}$ have at least two distinct values. Let them be $B_{i,\alpha_1}, ..., B_{i,\alpha_k}$ for some $\alpha_1 < ... < \alpha_k$ and $k \geq r$. Then, letting $\max C_{i,\alpha_j}$ and $\min C_{i,\alpha_j}$ denote the largest and smallest number in $B_{i,\alpha_j} \cap C$.

$$\sum_{j=1}^{k} \left(\max B_{i,\alpha_j} - \min B_{i,\alpha_j}\right) \geq \sum_{j=1}^{k} 1 = k \geq r$$

On the other hand, we have

$$\sum_{j=1}^{k} \left(\max B_{i,\alpha_j} - \min B_{i,\alpha_j}\right) = \max B_{i,\alpha_k} - \min B_{i,\alpha_1} + \sum_{i=1}^{k-1} \left(\max B_{i,\alpha_j} - \min B_{i,\alpha_{j+1}}\right) \leq r-1$$

which is contradiction. Thus, we must have $d(C) \leq m(r-1)$. ☐

Lemma 15. For each of f's MIPA C, there *exists* a pseudosolution $\mathcal{P}$ defined by a labeling $\mathcal{L}$, i.e. $\mathcal{P} = \mathit{pseudosol}(f, \mathcal{B}, \mathcal{L})$, such that $||C| - |\mathcal{P}|| \leq rmn/t$, where r, n, and t are given in Algorithm 4. With the parameters specified in Algorithm 4, the bound becomes $\epsilon mn/100$.

Proof. Construct $\mathcal{L}$ as follows.

1. If $B_{i,j} \cap \mathcal{P}$ consists of only one number, let $l_{i,j}$ be that number.

2. If $B_{i,j} \cap \mathcal{P}$ consists of two or more distinct numbers, choose $l_{i,j}$ arbitrarily from those numbers. Note that there are $d(C) \leq m(r-1)$ blocks of this category.

3. If $B_{i,j} \cap \mathcal{P}$ is empty, choose $l_{i,j}$ arbitrarily so that it does not violate the partial orders of those $l_{i,j}$'s already set.

Then, $\mathcal{P}$ and C only differ in those blocks in scenario 2 (at most $m(r-1)$ blocks due to Lemma 14, each of which has n/t elements. Therefore, we have $||C|-|\mathcal{P}|| \leq m(r-1)(n/t) \leq \epsilon mn/100$.

□

Lemma 16. There are at most $\binom{m+t}{m}^r$ unique pseudosolutions/labels.

Proof. Let $\mathcal{G}$ be a $m \times t$ grid. Say p is G's *diagonal path* iff p starts at $(0,0)$, moves in either rightward or upward direction (corresponding to horizontal and vertical lines), and ends at (m,t). Let $\mathcal{D}$ be the set of all diagonal paths of $\mathcal{G}$. It is a well-known fact $|\mathcal{D}| = \binom{m+t}{m}$.

Denote $\mathcal{F}_L$ be the set of all labels (of some pseudosolutions).

Consider a mapping $\chi : \mathcal{F}_L \to \mathcal{D}^r$ such that $\chi(\mathcal{L}) = (p_1, ..., p_r)$ where p_k is a diagonal path of $\mathcal{G}$ determined as follows (see Figure 5-1 for demonstration.)

1. For row i, put a vertical line at the position j which $\mathcal{L}_{i,j} < k$ and $\mathcal{L}_{i,j+1} \geq k$.

2. Complete the diagonal path p_k with horizontal lines.

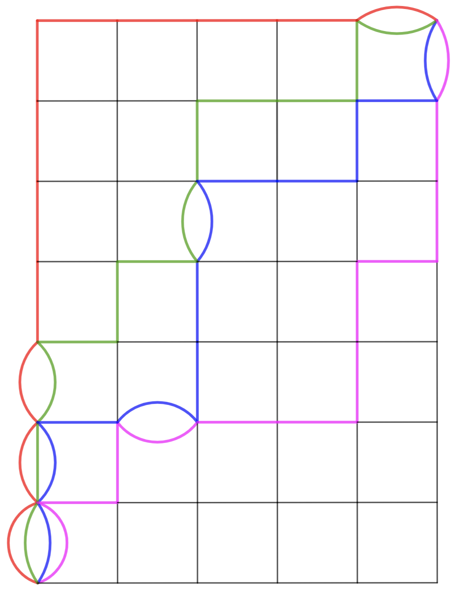
The monotonicity of $\mathcal{L}$ guarantees that the vertical lines appears in a left-to-right pattern; thus creating a valid diagonal path.

Now we show that χ is an injection by *decoding* $\mathcal{L}$ from $\chi(\mathcal{L}) = (p_1, ..., p_r)$ which can be done as follows.

1. Draw all r diagonal paths $p_1, ..., p_r$ on $\mathcal{G}$.

$$\mathcal{L} = \begin{pmatrix} 1 & 1 & 1 & 1 & 2 \\ 1 & 1 & 2 & 2 & 3 \\ 1 & 1 & 3 & 3 & 3 \\ 1 & 2 & 3 & 3 & 4 \\ 2 & 2 & 3 & 3 & 4 \\ 3 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 \end{pmatrix}$$

(a) Label $\mathcal{L}$ with dimension 7×5 whose entries are in $r = 4$ and in increasing order.



(b) $\chi(\mathcal{L}) = (p_1, p_2, p_3, p_4)$ drawn on a 7×5 grid labelled by red, green, blue, and pink diagonal paths respectively.

Figure 5-1: Demonstration of the mapping $\chi : \mathcal{F}_L \to \mathcal{D}^r$ with $m = 7, t = 5$, and $r = 4$.

2. Assign $\mathcal{L}_{i,j}$ to be the number of vertical lines that are in row i and in the left of $\mathcal{G}_{i,j}$.

The correctness of such decoding is justified by the following fact: if $\mathcal{L}_{i,j} = k$, then there are *exactly* k vertical lines that are in row i and in the left of $\mathcal{G}_{i,j}$ – one line from each of $p_1, ..., p_k$.

Therefore, χ is an injection which implies that the number of unique labels is $|\mathcal{F}_L| \leq |\mathcal{D}^r| = |\mathcal{D}|^r = \binom{m+t}{m}^r$. $\square$

Definition 17. Say a pseudosolution $\mathcal{P}$ of label $\mathcal{L}$ is *bad* (with respect to samples $\mathcal{S}$) iff $||\mathcal{P}| - \frac{n}{ts} \cdot agreement(\mathcal{B}, \mathcal{L}, \mathcal{S})| > \epsilon mn/100$. In other words, the agreement of $\mathcal{L}$ with the samples $\mathcal{S}$ does not approximate $\mathcal{P}$.

The following claim argues that any pseudosolution is bad with small probability.

Claim 18. For any pseudosolution $\mathcal{P}$, we have $\Pr(\mathcal{P} \text{ is bad}) \leq 2 \cdot \exp(-\epsilon^2 tms/30000)$. Furthermore, a bad pseudosolution exists with probability at most $2 \cdot \exp(-\epsilon^2 tms/30000)\binom{m+t}{m}^r$.

Proof. Notice that $agreement(\mathcal{B}, \mathcal{L}, \mathcal{S})$ is a sum of tms independent Bernoulli random variables. Furthermore, $\mathbb{E}(agreement(\mathcal{B}, \mathcal{L}, \mathcal{S})) = \frac{ts}{n} \cdot |\mathcal{P}|$. Therefore,

$$\begin{aligned}
\Pr(\mathcal{P} \text{ is bad}) &= \Pr(\left| |\mathcal{P}| - \frac{n}{ts} \cdot agreement(\mathcal{B}, \mathcal{L}, \mathcal{S}) \right| > \epsilon mn/100) \\
&= \Pr(|agreement(\mathcal{B}, \mathcal{L}, \mathcal{S}) - \mathbb{E}(agreement(\mathcal{B}, \mathcal{L}, \mathcal{S}))| > \epsilon tms/100) \\
&\leq 2 \cdot \exp(-\epsilon^2 tms/30000)
\end{aligned}$$

as wished. Finally, there are at most $\binom{m+t}{m}^r$ unique pseudosolutions (Lemma 16). The union bound gives the final part of the claim. □

Corollary 19. With the parameters specified in Algorithm 4, there is no bad pseudosolution with probability at least $1 - \gamma$.

Proof. The probability that a bad pseudosolution exists is at most

$$\begin{aligned}
2 \cdot \exp(-\epsilon^2 tms/30000) \binom{m+t}{m}^r &\leq 2 \cdot \exp(-\epsilon^2 tms/30000) \cdot \exp(rt \cdot \log e \left(1 + \frac{m}{t}\right)) \\
&= 2 \exp(rt \cdot \log e \left(1 + \frac{m}{t}\right) - \epsilon^2 tms/30000) \\
&= 2 \cdot \exp(-\log(2/\gamma)) = \gamma.
\end{aligned}$$

□

Claim 20. If there are no bad pseudosolutions, then $\left| \widetilde{L} - \mathsf{OPT}_{\mathsf{VAL}} \right| \leq \epsilon mn$.

Proof. The proof is identical to that of Theorem 8. The details of the proof is omitted for reasons of succinctness. □

5.1 Special Cases: Binary Arrays

In this section, we explore such algorithm in the special cases where we have additional assumptions on the array.

Corollary 21. Let A is a $m \times n$ 2-dimensional array $(m \leq n)$ whose entries are in $\{0, 1\}$. Let $\epsilon < 1$ be an additive error parameter. Then there is an algorithm that estimates $d_{\text{edit}}^{\text{mono}}(A)$ within ϵmn additive error with probability at least $1-\delta$ using $\Theta\left(\epsilon^{-3} \log(1 + \epsilon m) + \epsilon^2 \log \delta^{-1}\right)$ samples.

Proof. To see this, we change the parameters in Algorithm 4 with $r = 2$. The number of samples needed has become $\Theta\left(\epsilon^{-3} \log(1 + \epsilon m) + \epsilon^2 \log \delta^{-1}\right)$.

□

Chapter 6

Estimating L_1 Distance to Monotonicity in 1-Dimensional Sequence

Given two functions $f, g : [n] \to \mathbb{R}$. We define their L_1 distance $d_1(f, g)$ to be the smallest change in total magnitude needed to equate f and g. Formally, we define

$$d_1(f, g) = \sum_{i \in [n]} |f(i) - g(i)|.$$

We also define the L_1 distance to monotonicity of f to be the smallest L_1 distance to a monotone function. In particular, we can write

$$d_1^{\text{mono}}(f) = \min_{\substack{\text{monotone fn} \\ g:[n]\to\mathbb{R}}} d_1(f, g) = \min_{\substack{\text{monotone fn} \\ g:[n]\to\mathbb{R}}} \sum_{i \in [n]} |f(i) - g(i)|.$$

6.1 Estimating d_1^{mono} of a Discrete-Ranged Function

For this section, we consider a function $f : [n] \to [r]$. Define the following optimization problem.

Problem 22. Given a function $f : [n] \to [r]$. We want to approximate f's L_1 distance to its closest monotone function *whose range is contained in* $[r]$. Formally, we want to approximate

n	1	2	3	4	5	6	7	8	9	10
$f(n)$	5	1	2	3	3	1	4	2	5	4

(a) A sequence f with $n = 10$ and $r = 5$.

1	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	1	0	1	1
1	0	0	1	1	0	1	0	1	1
1	0	1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1	1	1

(b) Grid reduction of f, denoted Grid(f).

Figure 6-1: Shown above is a grid reduction from f (above) to Grid(f) (below). Notice that each column $i \in [n]$ of Grid(f) has 1s in its bottom $f(i)$ entries and 0s in its top $r - f(i)$ entries.

$$d_1^{\text{mono, int}}(f) = \min_{\substack{\text{monotone fn} \\ g:[n]\to[r]}} d_1(f, g) = \min_{\substack{\text{monotone fn} \\ g:[n]\to[r]}} \sum_{i\in[n]} |f(i) - g(i)|.$$

To do so, we view a 1-dimensional sequence $f : [n] \to [r]$ as a binary $r \times n$ array Grid$(f) \in \{0,1\}^{r\times n}$ and draw connection from varied distance functions. Formally, we define *grid reduction* as follows.

Definition 23 (Grid Reduction)**.** Given a a function $f : [n] \to [r]$. Let grid-reduction of f, denoted Grid(f), to be an $r \times n$ array of 0's and 1's such that

$$\text{Grid}(f)_{i,j} = 1 \text{ iff } j \geq r - f(i) + 1 \text{ and } 0 \text{ iff } j \leq r - f(i)\ .$$

Furthermore, Grid$(\cdot)$ is an bijective mapping from a family of sequences in $[r]^n$ to a family of binary arrays in $\{0,1\}^{r\times n}$. See Figure 6-1 for demonstration.

The following relates the L_1 distance of f and edit distance of Grid(f) in a nice way.

Theorem 24. For any function $f : [n] \to [r]$, we have $d_1^{\text{mono}}(f) = d_{\text{edit}}^{\text{mono}}(\text{Grid}(f))$.

The proof of this theorem follows directly from the following theorems.

Theorem 25. For any function $f : [n] \to [r]$, we have $d_1^{\text{mono, int}}(f) = d_{\text{edit}}^{\text{mono}}(\text{Grid}(f))$.

Proof. We first show that $d_1^{\text{mono, int}}(f) \geq d_{\text{edit}}^{\text{mono}}(\text{Grid}(f))$. To do so, let h be the closest integer-valued monotone function to f. Consider a monotone $0-1$ grid $\text{Grid}(h)$. Then,

$$d_{\text{edit}}^{\text{mono}}(\text{Grid}(f)) \leq d_{\text{edit}}(\text{Grid}(f), \text{Grid}(h)) = \sum_{i=1}^{n} |f(i) - h(i)| = d_1(f, g) = d_1^{\text{mono, int}}(f).$$

Now we will show that $d_{\text{edit}}^{\text{mono}}(\text{Grid}(f)) \geq d_1^{\text{mono, int}}(f)$. To do so, let $\mathcal{H}$ be the closest monotone $0-1$ grid to $\text{Grid}(f)$ in edit distance. Furthermore, let $h : [n] \to [r]$ which $h(j) = \sum_{i\in[r]} \mathcal{H}_{i,j}$ for any $j \in [r]$. We first notice that h is monotone and $\text{Grid}(h) = \mathcal{H}$. Furthermore, we have

$$\begin{aligned} d_1^{\text{mono, int}}(f) \leq d_1(f, h) &= \sum_{i\in[n]} |f(i) - h(i)| = \sum_{i\in[n]}\sum_{j\in[r]} 1_{\mathcal{H}_{i,j} \neq \text{Grid}(f)_{i,j}} \\ &= d_{\text{edit}}(\text{Grid}(f), \mathcal{H}) = d_{\text{edit}}^{\text{mono}}(\text{Grid}(f)) \end{aligned}$$

as wished. □

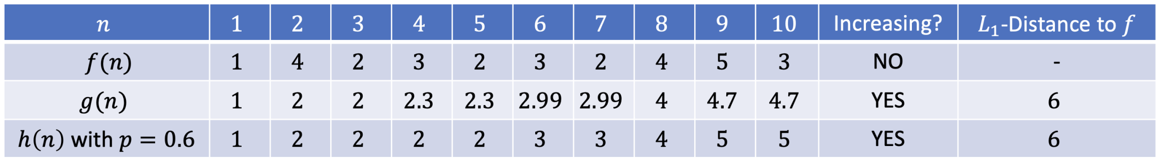
Theorem 26. For any function $f : [n] \to [r]$, we have $d_1^{\text{mono}}(f) = d_1^{\text{mono, int}}(f)$.

Proof. First of all, it is trivial that $d_1^{\text{mono}}(f) \leq d_1^{\text{mono, int}}(f)$. Thus, it suffices to show that $d_1^{\text{mono, int}}(f) \leq d_1^{\text{mono}}(f)$.

Let $g : [n] \to \mathbb{R}$ be the closest monotone function (w.r.t. L_1 distance) to f; that is g minimizes $\sum_{i\in[n]} |f(i) - g(i)|$. Denote $\{x\} = x - \lfloor x \rfloor$ to be the fractional part of x. Consider the following randomized process.

1. Uniformly choose $p \sim (0, 1)$.

2. Construct $h : [n] \to [r]$ such that $h(i) = \lfloor g(i) \rfloor$ if $\{g(i)\} < p$ and $\lceil g(i) \rceil$ otherwise.



n	1	2	3	4	5	6	7	8	9	10	Increasing?	L_1-Distance to f
$f(n)$	1	4	2	3	2	3	2	4	5	3	NO	-
$g(n)$	1	2	2	2.3	2.3	2.99	2.99	4	4.7	4.7	YES	6
$h(n)$ with $p = 0.6$	1	2	2	2	2	3	3	4	5	5	YES	6

Figure 6-2: Shown above is a 1-dimensional sequence $f : [10] \to [5]$ (row 1) with $n = 10$ and $r = 5$. The function $g : [10] \to \mathbb{R}$ (row 2) is *one of possibly many* of f's closest increasing functions with respect to L_1 distance with $d_1(f, g) = 6 := d_1^{\text{mono}}(f)$. The function $h : [10] \to [r]$ (row 3), generated via a randomized process with $p = 0.6$, is an increasing function who is expected to be at the same L_1 distance to f as g (in this case it is.) In other words, h minimizes $d_1(f, h^*)$ over range-$[r]$ functions h^*, which yields $d_1^{\text{mono, int}}(f) = d_1(f, h) = 6$. Theorem 26 states that $d_1^{\text{mono}}(f) = d_1^{\text{mono, int}}(f)$. In other words, we can optimize $d_1^{\text{mono}}(f)$ by looking up only those $h : [n] \to [r]$.

First, let's show that h is monotone. Consider $1 \leq i < j \leq n$. The monotonicity of g implies $g(i) \leq g(j)$ which implies $\lfloor g(i) \rfloor \leq \lfloor g(j) \rfloor$. If $\lfloor g(i) \rfloor \leq \lfloor g(j) \rfloor - 1$, then $h(i) \leq \lceil g(i) \rceil \leq \lfloor g(i) \rfloor + 1 \leq \lfloor g(j) \rfloor \leq h(j)$. Otherwise, $\lfloor g(i) \rfloor = \lfloor g(j) \rfloor$. We suppose, for contradiction, that $h(i) > h(j)$ (so h is not monotone). Note that this can only happen when $\lfloor g(i) \rfloor = \lfloor g(j) \rfloor$. Since $h(i) > h(j)$, we have that $h(i) = \lceil g(i) \rceil$ and $h(j) = \lfloor g(j) \rfloor$. In other words, we must have $p < \{g(i)\}$ and $p \geq \{g(j)\}$ which implies $\{g(i)\} > \{g(j)\}$. However, we then have $g(i) = \lfloor g(i) \rfloor + \{g(i)\} = \lfloor g(j) \rfloor + \{g(i)\} > \lfloor g(j) \rfloor + \{g(j)\} = g(j)$, resulting in a contradiction with the monotonicity of g.

Next, we will show that $\mathbb{E}\left[d_1(f, h)\right] = d_1(f, g)$. To show this, it suffices to show that for arbitrary $i \in [n]$, we have $\mathbb{E}\left(|f(i) - h(i)|\right) = |f(i) - g(i)|$, then use linearity of expectation. To show it, notice that if $g(i) \in \mathbb{Z}$, then $h(i) = g(i)$ so we are done. Else, let $g(i) = x + y$ where $x = \lfloor g(i) \rfloor$ and $y = \{g(i)\} \in (0, 1)$. Then, $\mathbb{E}\left(|f(i) - h(i)|\right) = y \cdot |f(i) - (x + 1)| + (1 - y) \cdot |f(i) - x|$. We will show that it is equal to $|f(i) - g(i)|$:

Case 1: $f(i) \leq x$. Consequently, we have $f(i) \leq x \leq g(i)$. Then,

$$\begin{aligned}
\mathbb{E}\left(|f(i)-h(i)|\right) &= y \cdot |f(i)-(x+1)| + (1-y) \cdot |f(i)-x| \\
&= y \cdot ((x+1)-f(i)) + (1-y)(x-f(i)) \\
&= x+y-f(i) \\
&= g(i)-f(i) \\
&= |f(i)-g(i)|.
\end{aligned}$$

Case 2: $f(i) \geq x+1$. Consequently, we have $f(i) \geq x+1 \geq g(i)$. Then,

$$\begin{aligned}
\mathbb{E}\left(|f(i)-h(i)|\right) &= y \cdot |f(i)-(x+1)| + (1-y) \cdot |f(i)-x| \\
&= y \cdot (f(i)-(x+1)) + (1-y) \cdot (f(i)-x) \\
&= f(i)-(x+y) \\
&= f(i)-g(i) \\
&= |f(i)-g(i)|.
\end{aligned}$$

As a result, we have shown that $\mathbb{E}\left[d_1(f,h)\right] = d_1(f,g) = d_1^{\text{mono}}(f)$. This means there must exists a monotone function $h^* : [n] \to [r]$ such that $d_1(f,h^*) \leq d_1^{\text{mono}}(f)$. Furthermore, we know that $d_1^{\text{mono,int}}(f) \leq d_1(f,h^*)$ so we have $d_1^{\text{mono,int}}(f) \leq d_1^{\text{mono}}(f)$ as wished. $\square$

Theorem 27. Let $f : [n] \to [r]$. Then there is an algorithm that estimates $d_1^{\text{mono}}(f)$ within ϵn additive error with probability at least $1-\delta$ using $\Theta(r^3\epsilon^{-3}\log(1+\epsilon) + r^2\epsilon^{-2}\log\delta^{-1})$ samples.

Proof. Construct a Grid(f) of dimension $r \times n$. Due to Theorem 24, it suffices to estimate $d_{\text{edit}}^{\text{mono}}(\text{Grid}(f))$, which can be done in $\Theta(r^3\epsilon^{-3}\log(1+\epsilon))$ queries via Theorem 12. We make a remark that a query to an entry (i,j) of Grid(f) is equivalent to querying $f(j)$ and then checking whether $f(j) \leq n-i+1$. $\square$

6.2 Estimating d_1^{mono} of a Continuous-Ranged Function

In this section, we generalize the range of f; from r discrete (consecutive) values $\{1, .., r\}$ to a continuous range $[0, r]$. We will construct a new function, based on f, whose range is discrete and has a small number of unique elements with hopes that the L_1 distance to monotonicity of such function is close to that of f. In particular, we propose the following theorem.

Theorem 28. Given $f : [n] \to [0, r]$ and $k \in \mathbb{R}^+$. Denote $f_k : [n] \to \{0, k, 2k, ..., \lfloor r/k \rfloor \cdot k\}$ such that $f_k(i) = k \cdot \lfloor f(i)/k \rfloor$. Then, we have $|d_1^{\text{mono}}(f) - d_1^{\text{mono}}(f_k)| \leq 2kn$. Furthermore, we can approximate $d_1^{\text{mono}}(f_k)$ within $\pm\epsilon n$ error with probability at least $1 - \delta$ using $\Theta\left(r^3\epsilon^{-3}\log\left(1 + \epsilon/k\right) + r^2\epsilon^{-2}\log\delta^{-1}\right)$ samples.

Proof. Let g and f_k^* be the f's and f_k's closest monotone function with respect to L_1 distance. That is $g = \arg\min_{\text{monotone } h} d_1(f, h)$ and $f_k^* = \arg\min_{\text{monotone } h} d_1(f_k^*, h)$. Finally, denote $g_k : [n] \to k \cdot [\lfloor r/k \rfloor]$ such that $g_k(i) = k \cdot \lfloor g(i)/k \rfloor$.

We first notice that $d_1(f, f_k) \leq kn$ and $d_1(g, g_k) \leq kn$ because $0 \leq f(i) - f_k(i) < k$ and $0 \leq g(i) - g(i) < k$ for any $i \in [n]$. Then,

$$\begin{aligned} d_1^{\text{mono}}(f) = d_1(f, g) &\geq d_1(f_k, g_k) - d_1(f, f_k) - d_1(g, g_k) \\ &\geq d_1(f_k, f_k^*) - kn - kn \\ &= d_1^{\text{mono}}(f_k) - 2kn. \end{aligned}$$

Moreover,

$$\begin{aligned} d_1^{\text{mono}}(f_k) = d_1(f_k, f_k^*) &\geq d_1(f, f_k^*) - d_1(f, f_k) \\ &\geq d_1(f, g) - kn \\ &= d_1^{\text{mono}}(f) - kn. \end{aligned}$$

This concludes that $|d_1^{\text{mono}}(f) - d_1^{\text{mono}}(f_k)| \leq 2kn$ as wished.

Finally, we can approximate f_k within $\pm\epsilon n$ by approximating f_k/k within $\pm\epsilon n/k$. Plus, f_k/k has a small and discrete range $\{0, 1, 2, ..., \lfloor r/k \rfloor\}$ which allows us to approximate its L_1 distance to monotonicity within an additive error $\pm\epsilon n/k$ in $\Theta\left(r^3\epsilon^{-3}\log\left(1+\epsilon/k\right)+r^2\epsilon^{-2}\log\delta^{-1}\right)$ samples by Theorem 12.

$\square$

Theorem 29. Given $f : [n] \to [0, r]$. Then there is an algorithm that estimates $d_1^{\mathrm{mono}(f)}$ within ϵn additive error with probability at least $1-\delta$ using $\Theta(r^3\epsilon^{-3}+r^2\epsilon^{-2}\log\delta^{-1})$ samples. The algorithm succeeds with at least constant probability, though can be easily amplified to $1-\delta$.

Proof. Set $k = \epsilon/8$. Theorem 28 says $|d_1^{\mathrm{mono}}(f) - d_1^{\mathrm{mono}}(f_k)| \le \epsilon n/4$. Furthermore, we can approximate $d_1^{\mathrm{mono}}(f_k)$ within an additive error $\pm\epsilon n/4$ using $\Theta(r^3\epsilon^{-3}+r^2\epsilon^{-2}\log\delta^{-1})$ samples, which in turn is within an additive error $\pm\epsilon n/2$ from $d_1^{\mathrm{mono}}(f)$.

$\square$

Bibliography

[1] Nir Ailon, Bernard Chazelle, Seshadhri Comandur, and Ding Liu. Estimating the distance to a monotone function. *Random Structures & Algorithms*, 31(3):371–383, 2007.

[2] Yevgeniy Dodis, Oded Goldreich, Eric Lehman, Sofya Raskhodnikova, Dana Ron, and Alex Samorodnitsky. Improved testing algorithms for monotonicity. In *Randomization, Approximation, and Combinatorial Optimization. Algorithms and Techniques*, pages 97–108. Springer, 1999.

[3] Michael L Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975.

[4] Michael Mitzenmacher and Saeed Seddighin. Improved sublinear time algorithm for longest increasing subsequence. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1934–1947. SIAM, 2021.

[5] Ilan Newman and Nithin Varma. New sublinear algorithms and lower bounds for lis estimation, 2021.

[6] Aviad Rubinstein, Saeed Seddighin, Zhao Song, and Xiaorui Sun. Approximation algorithms for lcs and lis with truly improved running times. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1121–1145. IEEE, 2019.

[7] Michael Saks and C Seshadhri. Estimating the longest increasing sequence in polylogarithmic time. *SIAM Journal on Computing*, 46(2):774–823, 2017.