Space-Efficient Algorithms for Longest Increasing Subsequence

Masashi Kiyomi

Yokohama City University. Yokohama, Japan
masashi@yokohama-cu.ac.jp

Hirotaka Ono

Nagoya University. Nagoya, Japan
ono@nagoya-u.jp

Yota Otachi

Kumamoto University. Kumamoto, Japan
otachi@cs.kumamoto-u.ac.jp

Pascal Schweitzer

TU Kaiserslautern. Kaiserslautern, Germany
schweitzer@cs.uni-kl.de

Jun Tarui

The University of Electro-Communications. Chofu, Japan
tarui@ice.uec.ac.jp

───── **Abstract** ─────

Given a sequence of integers, we want to find a longest increasing subsequence of the sequence. It is known that this problem can be solved in $O(n \log n)$ time and space. Our goal in this paper is to reduce the space consumption while keeping the time complexity small. For $\sqrt{n} \le s \le n$, we present algorithms that use $O(s \log n)$ bits and $O(\frac{1}{s} \cdot n^2 \cdot \log n)$ time for computing the length of a longest increasing subsequence, and $O(\frac{1}{s} \cdot n^2 \cdot \log^2 n)$ time for finding an actual subsequence. We also show that the time complexity of our algorithms is optimal up to polylogarithmic factors in the framework of sequential access algorithms with the prescribed amount of space.

1 Introduction

Given a sequence of integers (possibly with repetitions), the problem of finding a longest increasing subsequence (LIS, for short) is a classic problem in computer science which has many application areas including bioinfomatics and physics (see [38] and the references therein). It is known that LIS admits an $O(n \log n)$-time algorithm that uses $O(n \log n)$ bits of working space [37, 17, 2], where n is the length of the sequence.

35th Symposium on Theoretical Aspects of Computer Science (STACS 2018).
Editors: Rolf Niedermeier and Brigitte Vallée; Article No. 44; pp. 44:1–44:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A wide-spread algorithm achieving these bounds is PATIENCE SORTING, devised by Mallows [24, 25, 26]. Given a sequence of length n, PATIENCE SORTING partitions the elements of the sequence into so-called *piles*.

It can be shown that the number of piles coincides with the length of a longest increasing subsequence (see Section 3 for details). Combinatorial and statistical properties of the piles in PATIENCE SORTING are well studied (see [2, 8, 33]).

However, with the dramatic increase of the typical data sizes in applications over the last decade, a main memory consumption in the order of $\Theta(n \log n)$ bits is excessive in many algorithmic contexts, especially for basic subroutines such as LIS. We therefore investigate the existence of space-efficient algorithms for LIS.

Our results.   In this paper, we present the first space-efficient algorithms for LIS that are exact. We start by observing that when the input is restricted to permutations, an algorithm using $O(n)$ bits can be obtained straightforwardly by modifying a previously known algorithm (see Section 3.3). Next, we observe that a Savitch type algorithm [36] for this problem uses $O(\log^2 n)$ bits and thus runs in quasipolynomial time. However, we are mainly interested in space-efficient algorithms that also behave well with regard to running time. To this end we develop an algorithm that determines the length of a longest increasing subsequence using $O(\sqrt{n} \log n)$ bits which runs in $O(n^{1.5} \log n)$ time. Since the constants hidden in the O-notation are negligible, the algorithm, when executed in the main memory of a standard computer, may handle a peta-byte input on external storage.

More versatile, in fact, our space-efficient algorithm is *memory-adjustable* in the following sense. (See [3] for information on memory-adjustable algorithms.) When a memory bound s with $\sqrt{n} \leq s \leq n$ is given to the algorithm, it computes with $O(s \log n)$ bits of working space in $O(\frac{1}{s} \cdot n^2 \log n)$ time the length of a longest increasing subsequence. When $s = n$ our algorithm is equivalent to the previously known algorithms mentioned above. When $s = \sqrt{n}$ it uses, as claimed above, $O(\sqrt{n} \log n)$ bits and runs in $O(n^{1.5} \log n)$ time.

The algorithm only determines the length of a longest increasing subsequence. To actually find such a longest increasing subsequence, one can run the length-determining algorithm n times to successively construct the sought-after subsequence. This would give us a running time of $O(\frac{1}{s} \cdot n^3 \log n)$. However, we show that one can do much better, achieving a running time of $O(\frac{1}{s} \cdot n^2 \log^2 n)$ without any increase in space complexity, by recursively finding a *near-mid* element of a longest increasing subsequence.

To design the algorithms, we study the structure of the piles arising in PATIENCE SORTING in depth and show that maintaining certain information regarding the piles suffices to simulate the algorithm. Roughly speaking, our algorithm divides the execution of PATIENCE SORTING into $O(n/s)$ phases, and in each phase it computes in $O(n \log n)$ time information on the next $O(s)$ piles, while forgetting previous information.

Finally, we complement our algorithm with a lower bound in a restricted computational model. In the *sequential access model*, an algorithm can access the input only sequentially. We also consider further restricted algorithms in the *multi-pass model*, where an algorithm has to read the input sequentially from left to right and can repeat this multiple (not necessarily a constant number of) times. Our algorithm for the length works within the multi-pass model, while the one for finding a subsequence is a sequential access algorithm. Such algorithms are useful when large data is placed in an external storage that supports efficient sequential access. We show that the time complexity of our algorithms is optimal up to polylogarithmic factors in these models.

Related work.   The problem of finding a longest increasing subsequence (LIS) is among the most basic algorithmic problems on integer arrays and has been studied continuously since the early 1960's. It is known that LIS can be solved in $O(n \log n)$ time and space [37, 17, 2], and that any comparison-based algorithm needs $\Omega(n \log n)$ comparisons even for computing the length of a longest increasing subsequence [17, 32]. For the special case of LIS where the input is restricted to permutations, there are $O(n \log \log n)$-time algorithms [20, 6, 12]. PATIENCE SORTING, an efficient algorithm for LIS, has been a research topic in itself, especially in the context of Young tableaux [24, 25, 26, 2, 8, 33].

Recently, LIS has been studied intensively in the data-streaming model, where the input can be read only once (or a constant number of times) sequentially from left to right. This line of research was initiated by Liben-Nowell, Vee, and Zhu [22], who presented an exact one-pass algorithm and a lower bound for such algorithms. Their results were then improved and extended by many other groups [19, 38, 18, 34, 15, 28, 35]. These results give a deep understanding on streaming algorithms with a constant number of passes even under the settings with randomization and approximation. (For details on these models, see the very recent paper by Saks and Seshadhri [35] and the references therein.) On the other hand, multi-pass algorithms with a non-constant number of passes have not been studied for LIS.

While space-limited algorithms on both RAM and multi-pass models for basic problems have been studied since the early stage of algorithm theory, research in this field has recently intensified. Besides LIS, other frequently studied problems include sorting and selection [27, 7, 16, 30], graph searching [4, 14, 31, 9], geometric computation [10, 13, 5, 1], and k-SUM [39, 23].

2   Preliminaries

Let $\tau = \langle \tau(1), \tau(2), \ldots, \tau(n) \rangle$ be a sequence of n integers possibly with repetitions. For $1 \leq i_1 < \cdots < i_\ell \leq n$, the *subsequence* $\tau[i_1, \ldots, i_\ell]$ of τ is the sequence $\langle \tau(i_1), \ldots, \tau(i_\ell) \rangle$. A subsequence $\tau[i_1, \ldots, i_\ell]$ is an *increasing subsequence* of τ if $\tau(i_1) < \cdots < \tau(i_\ell)$. If $\tau(i_1) \leq \cdots \leq \tau(i_\ell)$, then the sequence τ is *non-decreasing*. We analogously define *decreasing subsequences* and *non-increasing subsequences*. By $\mathsf{lis}(\tau)$, we denote the length of a longest increasing subsequence of τ.

For example, consider a sequence $\tau_1 = \langle 2, 8, 4, 9, 5, 1, 7, 6, 3 \rangle$. It has an increasing subsequence $\tau_1[1, 3, 5, 8] = \langle 2, 4, 5, 6 \rangle$. Since there is no increasing subsequence of τ_1 with length 5 or more, we have $\mathsf{lis}(\tau_1) = 4$.

In the computational model in this paper, we use the RAM model with the following restrictions that are standard in the context of sublinear space algorithms. The input is in a read-only memory and the output must be produced on a write-only memory. We can use an additional memory that is readable and writable. Our goal is to minimize the size of the additional memory while keeping the running time fast. We measure space consumption in the number of bits used (instead of words) within the additional memory.

3   Patience Sorting

Since our algorithms are based on the classic PATIENCE SORTING, we start by describing it in detail and recalling some important properties regarding its internal configurations.

Internally, the algorithm maintains a collection of piles. A *pile* is a stack of integers. It is equipped with the procedures push and top: the push procedure appends a new element to become the new top of the pile; and the top procedure simply returns the element on top of

---

Algorithm 1 PATIENCE SORTING.

1: set $\ell := 0$ and initialize the dummy pile P_0 with the single element $-\infty$
2: **for** $i = 1$ **to** n **do**
3:     **if** $\tau(i) > \text{top}(P_\ell)$ **then**
4:         increment ℓ, let P_ℓ be a new empty pile, and set $j := \ell$
5:     **else**
6:         set j to be the smallest index with $\tau(i) \leq \text{top}(P_j)$
7:     push $\tau(i)$ to P_j
8: **return** ℓ

---

the pile, which is always the one that was added last.

We describe how PATIENCE SORTING computes $\text{lis}(\tau)$. See Algorithm 1. The algorithm scans the input τ from left to right (Line 2). It tries to push each newly read element $\tau(i)$ to a pile with a top element larger than or equal to $\tau(i)$. If on the one hand there is no such a pile, PATIENCE SORTING creates a new pile to which it pushes $\tau(i)$ (Line 4). On the other hand, if at least one such pile exists, PATIENCE SORTING pushes $\tau(i)$ to the oldest pile that satisfies the property (Line 6). After the scan, the number of piles is the output, which happens to be equal to $\text{lis}(\tau)$ (Line 8).

We return to the sequence $\tau_1 = \langle 2, 8, 4, 9, 5, 1, 7, 6, 3 \rangle$ for an example. The following illustration shows the execution of Algorithm 1 on τ_1. In each step the bold number is the newly added element. The colored (and underlined) elements in the final piles form a longest increasing subsequence $\tau_1[1, 3, 5, 8] = \langle 2, 4, 5, 6 \rangle$, which can be extracted as described below.

$$
\begin{array}{ccccccccc}
 & & & & & & & & \mathbf{3} \\
 & & \mathbf{4} & 4 & 4\ \mathbf{5} & \mathbf{1}\ 4\ 5 & 1\ 4\ 5 & 1\ 4\ 5\ \mathbf{6} & 1\ \underline{4}\ \underline{5}\ \underline{6} \\
\mathbf{2} & 2\ \mathbf{8} & 2\ 8 & 2\ 8\ \mathbf{9} & 2\ 8\ 9 & 2\ 8\ 9 & 2\ 8\ 9\ \mathbf{7} & 2\ 8\ 9\ 7 & \underline{2}\ 8\ 9\ 7 \\
\hline
P_1 & P_1\ P_2 & P_1\ P_2 & P_1\ P_2\ P_3 & P_1\ P_2\ P_3 & P_1\ P_2\ P_3 & P_1\ P_2\ P_3\ P_4 & P_1\ P_2\ P_3\ P_4 & P_1\ P_2\ P_3\ P_4
\end{array}
$$

▶ **Proposition 3.1** ([37, 17, 2]). *Given a sequence τ of length n, PATIENCE SORTING computes $\text{lis}(\tau)$ in $O(n \log n)$ time using $O(n \log n)$ bits of working space.*

3.1 Correctness of Patience Sorting

It is observed in [8] that when the input is a permutation π, the elements of each pile form a decreasing subsequence of π. This observation easily generalizes as follows.

▶ **Observation 3.2.** *Given a sequence τ, the elements of each pile constructed by PATIENCE SORTING form a non-increasing subsequence of τ.*

Hence, any increasing subsequence of τ can contain at most one element in each pile. This implies that $\text{lis}(\tau) \leq \ell$.

Now we show that $\text{lis}(\tau) \geq \ell$. Using the piles, we can obtain an increasing subsequence of length ℓ, in reversed order, as follows [2]:
1. Pick an arbitrary element of P_ℓ;
2. For $1 \leq i < \ell$, let $\tau(h)$ be the element picked from P_{i+1}. Pick the element $\tau(h')$ that was the top element of P_i when $\tau(h)$ was pushed to P_{i+1}.
Since $h' < h$ and $\tau(h') < \tau(h)$ in each iteration, the ℓ elements that are selected form an increasing subsequence of τ. This completes the correctness proof for PATIENCE SORTING.

The proof above can be generalized to show the following characterization for the piles.

Algorithm 2 Computing $\mathsf{lis}(\pi)$ with $O(n)$ bits and in $O(n^2)$ time.

1: set $\ell := 0$ and mark all elements in π as "unused"
2: **while** there is an "unused" element in π **do**
3:     increment ℓ and set $t := \infty$
4:     **for** $i = 1$ **to** n **do**   ▷ this for-loop constructs the next pile implicitly
5:         **if** $\pi(i)$ is unused and $\pi(i) < t$ **then**
6:             mark $\pi(i)$ as "used" and set $t := \pi(i)$ ▷ t is currently on top of P_ℓ
7: **return** ℓ

▶ **Proposition 3.3** ([8]). *$\tau(i) \in P_j$ if and only if a longest increasing subsequence of τ ending at $\tau(i)$ has length j.*

3.2 Time and space complexity of Patience Sorting

Observe that at any point in time, the top elements of the piles are ordered increasingly from left to right. Namely, $\mathtt{top}(P_k) < \mathtt{top}(P_{k'})$ if $k < k'$. This is observed in [8] for inputs with no repeated elements. We can see that the statement holds also for inputs with repetitions.

▶ **Observation 3.4.** *At any point in time during the execution of* PATIENCE SORTING *and for any k and k' with $1 \le k < k' \le \ell$, we have $\mathtt{top}(P_k) < \mathtt{top}(P_{k'})$ if P_k and $P_{k'}$ are nonempty.*

The observation above implies that Line 6 of Algorithm 1 can be executed in $O(\log n)$ time by using binary search. Hence, PATIENCE SORTING runs in $O(n \log n)$ time.

   The total number of elements in the piles is $O(n)$ and thus PATIENCE SORTING consumes $O(n \log n)$ bits. If it maintains all elements in the piles, it can compute an actual longest increasing subsequence in the same time and space complexity as described above. Note that to compute $\mathsf{lis}(\tau)$, it suffices to remember the top elements of the piles. However, the algorithm still uses $\Omega(n \log n)$ bits when $\mathsf{lis}(\tau) \in \Omega(n)$.

3.3 A simple $O(n)$-bits algorithm

Here we observe that, when the input is a permutation π of $\{1, \ldots, n\}$, $\mathsf{lis}(\pi)$ can be computed in $O(n^2)$ time with $O(n)$ bits of working space. The algorithm maintains a used/unused flag for each number in $\{1, \ldots, n\}$. Hence, this noncomparison-based algorithm cannot be generalized for general inputs directly.

   Let τ be a sequence of integers without repetitions. A subsequence $\tau[i_1, \ldots, i_\ell]$ is the *left-to-right minima subsequence* if $\{i_1, \ldots, i_\ell\} = \{i : \tau(i) = \min\{\tau(j) : 1 \le j \le i\}\}$. In other words, the left-to-right minima subsequence is made by scanning τ from left to right and greedily picking elements to construct a maximal decreasing subsequence.

   Burstein and Lankham [8, Lemma 2.9] showed that the first pile P_1 is the left-to-right minima subsequence of π and that the ith pile P_i is the left-to-right minima subsequence of a sequence obtained from π by removing all elements in the previous piles $P_1, \ldots, P_{i-1}$.

   Algorithm 2 uses this characterization of piles. The correctness follows directly from the characterization. It uses a constant number of pointers of $O(\log n)$ bits and a Boolean table of length n for maintaining "used" and "unused" flags. Thus it uses $n + O(\log n)$ bits working space in total. The running time is $O(n^2)$: each for-loop takes $O(n)$ time and the loop is repeated at most n times.

4 An algorithm for computing the length

In this section, we present our main algorithm that computes $\mathsf{lis}(\tau)$ with $O(s \log n)$ bits in $O(\frac{1}{s} \cdot n^2 \log n)$ time for $\sqrt{n} \leq s \leq n$. Note that the algorithm here outputs the length $\mathsf{lis}(\tau)$ only. The next section discusses efficient solutions to actually compute a longest sequence.

In the following, by P_i for some i we mean the ith pile obtained by (completely) executing PATIENCE SORTING unless otherwise stated. (We sometimes refer to a pile at some specific point of the execution.) Also, by $P_i(j)$ for $1 \leq j \leq |P_i|$ we denote the jth element added to P_i. That is, $P_i(1)$ is the first element added to P_i and $P_i(|P_i|)$ is the top element of P_i.

To avoid mixing up repeated elements, we assume that each element $\tau(j)$ of the piles is stored with its index j. In the following, we mean by "$\tau(j)$ is in P_i" that the jth element of τ is pushed to P_i. Also, by "$\tau(j)$ is $P_i(r)$" we mean that the jth element of τ is the rth element of P_i.

We start with an overview of our algorithm. It scans over the input $O(n/s)$ times. In each pass, it assumes that a pile P_i with at most s elements is given, which has been computed in the previous pass. Using this pile P_i, it filters out the elements in the previous piles $P_1, \ldots, P_{i-1}$. It then basically simulates PATIENCE SORTING but only in order to compute the next $2s$ piles. As a result of the pass, it computes a new pile P_j with at most s elements such that $j \geq i + s$.

The following observation, that follows directly from the definition of PATIENCE SORTING and Observation 3.4, will be useful for the purpose of filtering out elements in irrelevant piles.

▶ **Observation 4.1.** *Let $\tau(y) \in P_j$ with $j \neq i$. If $\tau(x)$ was the top element of P_i when $\tau(y)$ was pushed to P_j, then $j < i$ if $\tau(y) < \tau(x)$, and $j > i$ if $\tau(y) > \tau(x)$.*

Using Observation 4.1, we can obtain the following algorithmic lemma that plays an important role in the main algorithm.

▶ **Lemma 4.2.** *Having stored P_i explicitly in the additional memory and given an index $j > i$, the size $|P_k|$ for all $i + 1 \leq k \leq \min\{j, \mathsf{lis}(\tau)\}$ can be computed in $O(n \log n)$ time with $O((|P_i| + j - i) \log n)$ bits. If $\mathsf{lis}(\tau) < j$, then we can compute $\mathsf{lis}(\tau)$ in the same time and space complexity.*

Proof. Recall that PATIENCE SORTING scans the sequence τ from left to right and puts each element to the appropriate pile. We process the input in the same way except that we filter out, and thereby ignore, the elements in the piles P_h for which $h < i$ or $h > j$.

To this end, we use the following two filters whose correctness follows from Observation 4.1.

(Filtering P_h with $h < i$.) To filter out the elements that lie in P_h for some $h < i$, we maintain an index r that points to the element of P_i read most recently in the scan. Since P_i is given explicitly to the algorithm, we can maintain such a pointer r.

When we read a new element $\tau(x)$, we have three cases.

- If $\tau(x)$ is $P_i(r + 1)$, then we increment the index r.
- Else if $\tau(x) < P_i(r)$, then $\tau(x)$ is ignored since it is in P_h for some $h < i$.
- Otherwise we have $\tau(x) > P_i(r)$. In this case $\tau(x)$ is in P_h for some $h > i$.

(Filtering P_h with $h > j$.) The elements in P_h for $h > j$ can be filtered without maintaining additional information as follows. Let again $\tau(x)$ be the newly read element.

- If no part of P_j has been constructed yet, then $\tau(x)$ is in P_h for some $h \leq j$.
- Otherwise, we compare $\tau(x)$ and the element $\tau(y)$ currently on the top of P_j.
  - If $\tau(x) > \tau(y)$, then $\tau(x)$ is in P_h for some $h > j$, and thus ignored.
  - Otherwise $\tau(x)$ is in P_h for some $h \leq j$.

Algorithm 3 Computing $|P_k|$ for all k with $i + 1 \leq k \leq \min\{j, \mathsf{lis}(\tau)\}$ when P_i is given.

1: set $r := 0$ $\triangleright$ r points to the most recently read element in P_i
2: set $\ell := i$ $\triangleright$ the largest index of the piles constructed so far
3: initialize $p_{i+1}, \ldots, p_j$ to ∞ $\triangleright$ p_k is the element currently on top of P_k
4: initialize $c_{i+1}, \ldots, c_j$ to 0 $\triangleright$ c_k is the current size of P_k
5: **for** $x = 1$ **to** n **do**
       $\triangleright$ filtering out irrelevant elements
6:      **if** $\tau(x)$ is $P_i(r + 1)$ **then**
7:          increment r and continue the for-loop
8:      **else if** $\tau(x) < P_i(r)$ or $(\ell \geq j$ and $\tau(x) > p_j)$ **then**
9:          ignore the element and continue the for-loop
       $\triangleright$ push $\tau(x)$ to the appropriate pile
10:      **if** $\tau(x) > p_\ell$ **then**
11:          increment ℓ and set $h := \ell$
12:      **else**
13:          set h to be the smallest index with $\tau(i) < p_h$
14:      set $p_h := \tau(x)$ and increment c_h

We simulate PATIENCE SORTING only for the elements that pass both filters above. While doing so, we only maintain the top elements of the piles and additionally store the size of each pile. This requires at most $O((j - i) \log n)$ space, as required by the statement of the lemma. For details see Algorithm 3.

The running time remains the same since we only need constant number of additional steps for each step in PATIENCE SORTING to filter out irrelevant elements. If P_j is still empty after this process, we can conclude that $\mathsf{lis}(\tau)$ is the index of the newest pile constructed. ◄

The proof of Lemma 4.2 can be easily adapted to also compute the pile P_j explicitly. For this, we simply additionally store all elements of P_j as they are added to the pile.

▶ **Lemma 4.3.** *Given P_i and an index j such that $i < j \leq \mathsf{lis}(\tau)$, we can compute P_j in $O(n \log n)$ time with $O((|P_i| + |P_j| + j - i) \log n)$ bits.*

Assembling the lemmas of this section, we now present our first main result. The corresponding pseudocode of the algorithm can be found in Algorithm 4.

▶ **Theorem 4.4.** *There is an algorithm that, given an integer s satisfying $\sqrt{n} \leq s \leq n$ and a sequence τ of length n, computes $\mathsf{lis}(\tau)$ in $O(\frac{1}{s} \cdot n^2 \log n)$ time with $O(s \log n)$ bits of space.*

Proof. To apply Lemmas 4.2 and 4.3 at the beginning, we start with a dummy pile P_0 with a single dummy entry $P_0(1) = -\infty$. In the following, assume that for some $i \geq 0$ we computed the pile P_i of size at most s explicitly. We repeat the following process until we find $\mathsf{lis}(\tau)$.

In each iteration, we first compute the size $|P_k|$ for $i + 1 \leq k \leq i + 2s$. During this process, we may find $\mathsf{lis}(\tau) < i + 2s$. In such a case we output $\mathsf{lis}(\tau)$ and terminate. Otherwise, we find an index j such that $i + s + 1 \leq j \leq i + 2s$ and $|P_j| \leq n/s$. Since $s \geq \sqrt{n}$, it holds that $|P_j| \leq n/\sqrt{n} = \sqrt{n} \leq s$. We then compute P_j itself to replace i with j and repeat.

By Lemmas 4.2 and 4.3, each pass can be executed in $O(n \log n)$ time with $O(s \log n)$ bits. There are at most $\mathsf{lis}(\tau)/s$ iterations, since in each iteration the index i increases by at least s or $\mathsf{lis}(\tau)$ is determined. Since $\mathsf{lis}(\tau) \leq n$, the total running time is $O(\frac{1}{s} \cdot n^2 \log n)$. ◄

In the case of the smallest memory consumption we conclude the following corollary.

Algorithm 4 Computing $\mathsf{lis}(\tau)$ with $O(s \log n)$ bits in $O(\frac{1}{s} \cdot n^2 \log n)$ time.

1: set $i := 0$ and initialize the dummy pile P_0 with the single element $-\infty$
2: **loop**
3:      compute the size of P_k for all k with $i + 1 \leq k \leq i + 2s$
4:      **if** we find $\mathsf{lis}(\tau) < i + 2s$ **then**
5:          **return** $\mathsf{lis}(\tau)$
6:      let j be the largest index such that $|P_j| \leq s$ $\triangleright$ $i + s + 1 \leq j \leq i + 2s$
7:      compute P_j and set $i := j$

▶ **Corollary 4.5.** *Given a sequence τ of length n, $\mathsf{lis}(\tau)$ can be computed in $O(n^{1.5} \log n)$ time with $O(\sqrt{n} \log n)$ bits of space.*

5   An algorithm for finding a longest increasing subsequence

It is easy to modify the algorithm in the previous section in such a way that it outputs an element of the final pile $P_{\mathsf{lis}(\tau)}$, which is the last element of a longest increasing subsequence by Proposition 3.3. Thus we can repeat the modified algorithm n times (considering only the elements smaller than and appearing before the last output) and actually find a longest increasing subsequence.[1] The running time of this naïve approach is $O(\frac{1}{s} \cdot n^3 \log n)$.

As we claimed before, we can do much better. In fact, we need only an additional multiplicative factor of $O(\log n)$ instead of $O(n)$ in the running time, while keeping the space complexity as it is. In the rest of this section, we prove the following theorem.

▶ **Theorem 5.1.** *There is an algorithm that, given an integer s satisfying $\sqrt{n} \leq s \leq n$ and a sequence τ of length n, computes a longest increasing subsequence of τ in $O(\frac{1}{s} \cdot n^2 \log^2 n)$ time using $O(s \log n)$ bits of space.*

▶ **Corollary 5.2.** *Given a sequence τ of length n, a longest increasing subsequence of τ can be found in $O(n^{1.5} \log^2 n)$ time with $O(\sqrt{n} \log n)$ bits of space.*

We should point out that the algorithm in this section is *not* a multi-pass algorithm. However, we can easily transform it without any increase in the time and space complexity so that it works as a sequential access algorithm.

5.1   High-level idea

We first find an element that is in a longest increasing subsequence roughly in the middle. As we will argue, this can be done in $O(\frac{1}{s} \cdot n^2 \log n)$ time with $O(s \log n)$ bits by running the algorithm from the previous section twice, once in the ordinary then once in the reversed way. We then divide the input into the left and right parts at a near-mid element and recurse.

The space complexity remains the same and the time complexity increases only by an $O(\log n)$ multiplicative factor. The depth of recursion is $O(\log n)$ and at each level of recursion the total running time is $O(\frac{1}{s} \cdot n^2 \log n)$. To remember the path to the current recursion, we need some additional space, but it is bounded by $O(\log^2 n)$ bits.

---

[1] This algorithm outputs a longest increasing subsequence in the reversed order. One can access the input in the reversed order and find a longest *decreasing* subsequence to avoid this issue.

Algorithm 5 REVERSE PATIENCE SORTING.

1: set $\ell := 0$ and initialize the dummy pile Q_0 with the single element $+\infty$
2: **for** $i = n$ **to** 1 **do**
3:     **if** $\tau(i) < \texttt{top}(Q_\ell)$ **then**
4:         increment ℓ, let Q_ℓ to be a new empty pile, and set $j := \ell$
5:     **else**
6:         set j to be the smallest index with $\tau(i) > \texttt{top}(Q_j)$
7:     push $\tau(i)$ to Q_j
8: **return** ℓ

5.2 A subroutine for short longest increasing sequences

We first solve the base case in which $\mathsf{lis}(\tau) \in O(n/s)$. In this case, we use the original PATIENCE SORTING and repeat it $O(n/s)$ times. We present the following general form first.

▶ **Lemma 5.3.** *Let τ be a sequences of length n and $\mathsf{lis}(\tau) = k$. Then a longest increasing subsequence of τ can be found in $O(k \cdot n \log k)$ time with $O(k \log n)$ bits.*

Proof. Without changing the time and space complexity, we can modify the original PATIENCE SORTING so that

- it maintains only the top elements of the piles;
- it ignores the elements larger than or equal to a given upper bound; and
- it outputs an element in the final pile.

We run the modified algorithm $\mathsf{lis}(\tau)$ times. In the first run, we have no upper bound. In the succeeding runs, we set the upper bound to be the output of the previous run. In each run the input to the algorithm is the initial part of the sequence that ends right before the last output. The entire output forms a longest increasing sequence of τ.[2]

Since $\mathsf{lis}(\tau) = k$, modified PATIENCE SORTING maintains only k piles. Thus each run takes $O(n \log k)$ time and uses $O(k \log n)$ bits. The lemma follows since this is repeated k times and each round only stores $O(\log n)$ bits of information from the previous round. ◀

The following special form of the lemma above holds since $n/s \le s$ when $s \ge \sqrt{n}$.

▶ **Corollary 5.4.** *Let τ be a sequence of length n and $\mathsf{lis}(\tau) \in O(n/s)$ for some s with $\sqrt{n} \le s \le n$. A longest increasing subsequence of τ can be found in $O(\frac{1}{s} \cdot n^2 \log n)$ time with $O(s \log n)$ bits.*

5.3 A key lemma

As mentioned above, we use a reversed version of our algorithm. REVERSE PATIENCE SORTING is the reversed version of PATIENCE SORTING: it reads the input from right to left and uses the reversed inequalities. (See Algorithm 5.) REVERSE PATIENCE SORTING computes the length of a longest decreasing subsequence in the reversed sequence, which is a longest increasing subsequence in the original sequence. Since the difference between the two algorithms is small, we can easily modify our algorithm in Section 4 for the length so that it simulates REVERSE PATIENCE SORTING instead of PATIENCE SORTING.

---

[2] Again this output is reversed. We can also compute the output in nonreversed order as discussed before.

Let Q_i be the ith pile constructed by REVERSE PATIENCE SORTING as in Algorithm 5. Using Proposition 3.3, we can show that for each $\tau(i)$ in Q_j, the longest decreasing subsequence of the reversal of τ ending at $\tau(i)$ has length j. This is equivalent to the following observation.

▶ **Observation 5.5.** $\tau(i) \in Q_j$ if and only if a longest increasing subsequence of τ starting at $\tau(i)$ has length j.

This observation immediately gives the key lemma below.

▶ **Lemma 5.6.** $P_k \cap Q_{\mathsf{lis}(\tau)-k+1} \neq \emptyset$ for all k with $1 \leq k \leq \mathsf{lis}(\tau)$.

Proof. Let $\langle \tau(i_1), \ldots, \tau(i_\ell) \rangle$ be a longest increasing subsequence of τ. Proposition 3.3 implies that $\tau(i_k) \in P_k$. The subsequence $\langle \tau(i_k), \ldots, \tau(i_\ell) \rangle$ is a longest increasing subsequence of τ starting at $\tau(i_k)$ since otherwise $\langle \tau(i_1), \ldots, \tau(i_\ell) \rangle$ is not longest. Since the length of $\langle \tau(i_k), \ldots, \tau(i_\ell) \rangle$ is $i_\ell - k + 1 = \mathsf{lis}(\tau) - k + 1$, we have $\tau(k) \in Q_{\mathsf{lis}(\tau)-k+1}$. ◀

Note that the elements of P_k and $Q_{\mathsf{lis}(\tau)-k+1}$ are not the same in general. For example, by applying REVERSE PATIENCE SORTING to $\tau_1 = \langle 2, 8, 4, 9, 5, 1, 7, 6, 3 \rangle$, we get $Q_1 = \langle 3, 6, 7, 9 \rangle$, $Q_2 = \langle 1, 5, 8 \rangle$, $Q_3 = \langle 4 \rangle$, and $Q_4 = \langle 2 \rangle$ as below. (Recall that $P_1 = \langle 2, 1 \rangle$, $P_2 = \langle 8, 4, 3 \rangle$, $P_3 = \langle 9, 5 \rangle$, and $P_4 = \langle 7, 6 \rangle$.) The following diagram depicts the situation. The elements shared by P_k and $Q_{\mathsf{lis}(\tau)-k+1}$ are colored and underlined.

			9	9		9		9				3			
	7	7	7	7	7		7 **8**	_7_ 8				3			
6	6	6	6 **5**	6 5	6 5	6 5	_6_ _5_			1	_4_	_5_	_6_		
3	3	3	3 **1**	3 1	3 1 **4**	3 1 4	3 1 _4_ **2**	_2_	8	9	_7_				
Q_1	Q_1	Q_1	$Q_1\,Q_2$	$Q_1\,Q_2$	$Q_1\,Q_2\,Q_3$	$Q_1\,Q_2\,Q_3$	$Q_1\,Q_2\,Q_3\,Q_4$	$P_1\,P_2\,P_3\,P_4$							

5.4 The algorithm

We first explain the subroutine for finding a near-mid element in a longest increasing subsequence.

▶ **Lemma 5.7.** Let s be an integer satisfying $\sqrt{n} \leq s \leq n$. Given a sequence τ of length n, the kth element of a longest increasing subsequence of τ for some k with $\mathsf{lis}(\tau)/2 \leq k < \mathsf{lis}(\tau)/2 + n/s$ can be found in $O(\frac{1}{s} \cdot n^2 \log n)$ time using $O(s \log n)$ bits of space.

Proof. We slightly modify Algorithm 4 so that it finds an index k and outputs P_k such that $|P_k| \leq s$ and $\mathsf{lis}(\tau)/2 \leq k \leq \mathsf{lis}(\tau)/2 + n/s$. Such a k exists since the average of $|P_i|$ for $\mathsf{lis}(\tau)/2 \leq i < \mathsf{lis}(\tau)/2 + n/s$ is at most s. The time and space complexity of this phase are as required by the lemma.

We now find an element in $P_k \cap Q_{\mathsf{lis}(\tau)-k+1}$. Since the size $|Q_{\mathsf{lis}(\tau)-k+1}|$ is not bounded by $O(s)$ in general, we cannot store $Q_{\mathsf{lis}(\tau)-k+1}$ itself. Instead use the reversed version of the algorithm in Section 4 to enumerate it. Each time we find an element in $Q_{\mathsf{lis}(\tau)-k+1}$, we check whether it is included in P_k. This can be done with no loss in the running time since P_k is sorted and the elements of $Q_{\mathsf{lis}(\tau)-k+1}$ arrive in increasing order. ◀

The next technical but easy lemma allows us to split the input into two parts at an element of a longest increasing subsequence and to solve the smaller parts independently.

▶ **Lemma 5.8.** Let $\tau(j)$ be the kth element of a longest increasing subsequence of a sequence τ. Let τ_L be the subsequence of $\tau[1, \ldots, j-1]$ formed by the elements smaller than $\tau(j)$. Similarly let τ_R be the subsequence of $\tau[j + 1, \ldots, |\tau|]$ formed by the elements larger than $\tau(j)$. Then, a longest increasing subsequence of τ can be obtained by concatenating a longest increasing subsequence of τ_L, $\tau(j)$, and a longest increasing subsequence of τ_R, in this order.

Algorithm 6 Recursively finding a longest increasing subsequence of ρ.

1: RECURSIVELIS(ρ, $-\infty$, $+\infty$)
2: **procedure** RECURSIVELIS(τ, lb, ub)
3:     $\tau' :=$ the subsequence of τ formed by the elements $\tau(i)$ such that lb $< \tau(i) <$ ub
              ▷ τ' is not explicitly computed but provided by ignoring the irrelevant elements
4:     compute lis(τ')
5:     **if** lis(τ') $\leq 3|\tau'|/s$ **then**
6:         **output** a longest increasing subsequence of τ' ▷ Lemma 5.3
7:     **else**
8:         find the kth element $\tau'(j)$ of a longest increasing subsequence of τ'
                  for some k with lis(τ')$/2 \leq k <$ lis(τ')$/2 + |\tau'|/s$
9:         RECURSIVELIS($\tau'[1, \ldots, j-1]$, lb, τ'(j))
10:        **output** $\tau'(j)$
11:        RECURSIVELIS($\tau'[j+1, \ldots, |\tau'|]$, $\tau'(j)$, ub)

Proof. Observe that the concatenated sequence is an increasing subsequence of τ. Thus it suffices to show that lis(τ_L) + lis(τ_R) + 1 $\geq$ lis(τ). Let $\tau[i_1, \ldots, i_{\mathsf{lis}(\tau)}]$ be a longest increasing subsequence of τ such that $i_k = j$. From the definition, $\tau[i_1, \ldots, i_{k-1}]$ is a subsequence of τ_L, and $\tau[i_{k+1}, \ldots, i_{\mathsf{lis}(\tau)}]$ is a subsequence of τ_R. Hence lis(τ_L) $\geq k-1$ and lis(τ_R) $\geq$ lis(τ) $- k$, and thus lis(τ_L) + lis(τ_R) + 1 $\geq$ lis(τ). ◀

As Lemma 5.8 suggests, after finding a near-mid element $\tau(k)$, we recurse into τ_L and τ_R. If the input τ' to a recursive call has small lis(τ'), we directly compute a longest increasing subsequence. See Algorithm 6 for details of the whole algorithm. Correctness follows from Lemma 5.8 and correctness of the subroutines.

5.5 Time and space complexity

In Theorem 5.1, the claimed running time is $O(\frac{1}{s} \cdot n^2 \log^2 n)$. To prove this, we first show that the depth of the recursion is $O(\log n)$. We then show that the total running time in each recursion level is $O(\frac{1}{s} \cdot n^2 \log n)$. The claimed running time is guaranteed by these bounds.

▶ **Lemma 5.9.** *Given a sequence τ, the depth of the recursions invoked by* RECURSIVELIS *of Algorithm 6 is at most* $\log_{6/5}$ lis(τ')*, where τ' is the subsequence of τ computed in Line 3.*

Proof. We proceed by induction on lis(τ'). If lis(τ') $\leq 3|\tau'|/s$, then no recursive call occurs, and hence the lemma holds. In the following, we assume that lis(τ') $= \ell > 3|\tau'|/s$ and that the statement of the lemma is true for any sequence τ'' with lis(τ'') $< \ell$.

Since $\ell > 3|\tau'|/s$, we recurse into two branches on subsequences of τ'. From the definition of k in Line 8 of Algorithm 6, the length of a longest increasing subsequence is less than $\ell/2 + |\tau'|/s$ in each branch. Since $\ell/2 + |\tau'|/s < \ell/2 + \ell/3 = 5\ell/6$, each branch invokes recursions of depth at most $\log_{6/5}(5\ell/6) = \log_{6/5} \ell - 1$. Therefore the maximum depth of the recursions invoked by their parent is at most $\log_{6/5} \ell$. ◀

▶ **Lemma 5.10.** *Given a sequence τ of length n, the total running time at each depth of recursion excluding further recursive calls in Algorithm 6 takes* $O(\frac{1}{s} n^2 \log n)$ *time.*

Proof. In one recursion level, we have many calls of RECURSIVELIS on pairwise non-overlapping subsequences of τ. For each subsequence τ', the algorithm spends time $O(\frac{1}{s}|\tau'|^2 \log |\tau'|)$. Thus the total running time at a depth is $O(\sum_{\tau'} \frac{1}{s}|\tau'|^2 \log |\tau'|)$, which is $O(\frac{1}{s} n^2 \log n)$ since $\sum_{\tau'} |\tau'|^2 \leq |\tau|^2 = n^2$. ◀

Finally we consider the space complexity of Algorithm 6.

▶ **Lemma 5.11.** *Algorithm 6 uses $O(s \log n)$ bits of working space on sequences of length n.*

Proof. We have already shown that each subroutine uses $O(s \log n)$ bits. Moreover, this space of working memory can be discarded before another subroutine call occurs. Only a constant number of $O(\log n)$-bit words are passed to the new subroutine call. We additionally need to remember the stack trace of the recursion. The size of this additional information is bounded by $O(\log^2 n)$ bits since each recursive call is specified by a constant number of $O(\log n)$-bit words and the depth of recursion is $O(\log n)$ by Lemma 5.9. Since $\log^2 n \in O(s \log n)$ for $s \geq \sqrt{n}$, the lemma holds.   ◀

6   Lower bound for algorithms with sequential access

An algorithm is a *sequential access* algorithm if it can access elements in the input array only sequentially. In our situation this means that for a given sequence, accessing the ith element of the sequence directly after having accessed the jth element of the sequence costs time at least linear in $|i - j|$. As opposed to the RAM, any Turing machine in which the input is given on single read-only tape has this property. Note that any lower bound for sequential access algorithms in an asymptotic form is applicable to multi-pass algorithms as well since every multi-pass algorithm can be simulated by a sequential access algorithm with the same asymptotic behavior. Although some of our algorithms are not multi-pass algorithms, it is straightforward to transform them to sequential access algorithms with the same time and space complexity.

To show a lower bound on the running time of sequential access algorithms with limited working space, we need the concept of communication complexity (see [21] for more details). Let f be a function. Given $\alpha \in \mathcal{A}$ to the first player Alice and $\beta \in \mathcal{B}$ to the second player Bob, the players want to compute $f(\alpha, \beta)$ together by sending bits to each other (possibly multiple times). The communication complexity of f is the maximum number of bits transmitted between Alice and Bob over all inputs by the best protocol for f. Now consider the following variant of the LIS problem: Alice gets the first half of a permutation π of $\{1, \ldots, 2n\}$ and Bob gets the second half. They compute $\mathsf{lis}(\pi)$ together. It is known that this problem has $\Omega(n)$ communication complexity (even with 2-sided error randomization) [22, 19, 38].

For sequential access algorithms, we can show the following lower bound by using the communication complexity lower bound mentioned above.

▶ **Theorem 6.1.** *Given a permutation π of $\{1, \ldots, 4n\}$, any sequential access (possibly randomized) algorithm computing $\mathsf{lis}(\pi)$ using b bits takes $\Omega(n^2/b)$ time.*

7   Concluding remarks

Our result raises the following question: "Do $o(\sqrt{n})$-space polynomial-time algorithms for LIS exist?" An unconditional 'no' answer would be surprising as it implies $\mathrm{SC} \neq \mathrm{P} \cap \mathrm{PolyL}$, where SC (Steve's Class) is the class of problems that can be solved by an algorithm that simultaneously runs in polynomial-time and polylogarithmic-space [11, 29]. A possibly easier question asks for the existence of a log-space algorithm. For this question, one might be able to give some evidence for a 'no' answer by showing NL-hardness of (a decision version of) LIS.

As a final remark, we would like to mention some known results that have a mysterious coincidence in space complexity with our results. For $(1 + \epsilon)$-approximation of $\mathsf{lis}(\pi)$ by

one-pass streaming algorithms, it is known that $O(\sqrt{n/\epsilon} \cdot \log n)$ bits are sufficient [19] and $\Omega(\sqrt{n/\epsilon})$ bits are necessary [15, 18]. We were not able to find any connection here and do not claim anything concrete about this coincidence.

References

1   Hee-Kap Ahn, Nicola Baraldo, Eunjin Oh, and Francesco Silvestri. A time-space trade-off for triangulations of points in the plane. In Yixin Cao and Jianer Chen, editors, *Computing and Combinatorics - 23rd International Conference, COCOON 2017, Hong Kong, China, August 3-5, 2017, Proceedings*, volume 10392 of *Lecture Notes in Computer Science*, pages 3–12. Springer, 2017. `doi:10.1007/978-3-319-62389-4_1`.

2   David Aldous and Persi Diaconis. Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem. *Bulletin of the American Mathematical Society*, 36(4):413–432, 1999. `doi:10.1090/S0273-0979-99-00796-X`.

3   Tetsuo Asano, Amr Elmasry, and Jyrki Katajainen. Priority queues and sorting for read-only data. In T.-H. Hubert Chan, Lap Chi Lau, and Luca Trevisan, editors, *Theory and Applications of Models of Computation, 10th International Conference, TAMC 2013, Hong Kong, China, May 20-22, 2013. Proceedings*, volume 7876 of *Lecture Notes in Computer Science*, pages 32–41. Springer, 2013. `doi:10.1007/978-3-642-38236-9_4`.

4   Tetsuo Asano, Taisuke Izumi, Masashi Kiyomi, Matsuo Konagaya, Hirotaka Ono, Yota Otachi, Pascal Schweitzer, Jun Tarui, and Ryuhei Uehara. Depth-first search using o(n) bits. In Hee-Kap Ahn and Chan-Su Shin, editors, *Algorithms and Computation - 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*, volume 8889 of *Lecture Notes in Computer Science*, pages 553–564. Springer, 2014. `doi:10.1007/978-3-319-13075-0_44`.

5   Bahareh Banyassady, Matias Korman, Wolfgang Mulzer, André van Renssen, Marcel Roeloffzen, Paul Seiferth, and Yannik Stein. Improved time-space trade-offs for computing voronoi diagrams. In Heribert Vollmer and Brigitte Vallée, editors, *34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, March 8-11, 2017, Hannover, Germany*, volume 66 of *LIPIcs*, pages 9:1–9:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. `doi:10.4230/LIPIcs.STACS.2017.9`.

6   Sergei Bespamyatnikh and Michael Segal. Enumerating longest increasing subsequences and patience sorting. *Inf. Process. Lett.*, 76(1-2):7–11, 2000. `doi:10.1016/S0020-0190(00) 00124-1`.

7   Allan Borodin and Stephen A. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.*, 11(2):287–297, 1982. `doi:10.1137/0211022`.

8   Alexander Burstein and Isaiah Lankham. Combinatorics of patience sorting piles. *Séminaire Lotharingien de Combinatoire*, 54A:B54Ab, 2006. URL: `http://www.mat.univie.ac.at/~slc/wpapers/s54Aburlank.html`.

9   Sankardeep Chakraborty and Srinivasa Rao Satti. Space-efficient algorithms for maximum cardinality search, stack bfs, queue BFS and applications. In Yixin Cao and Jianer Chen, editors, *Computing and Combinatorics - 23rd International Conference, COCOON 2017, Hong Kong, China, August 3-5, 2017, Proceedings*, volume 10392 of *Lecture Notes in Computer Science*, pages 87–98. Springer, 2017. `doi:10.1007/978-3-319-62389-4_8`.

10   Timothy M. Chan and Eric Y. Chen. Multi-pass geometric algorithms. *Discrete & Computational Geometry*, 37(1):79–102, 2007. `doi:10.1007/s00454-006-1275-6`.

11   Stephen A. Cook. Deterministic cfl's are accepted simultaneously in polynomial time and log squared space. In Michael J. Fischer, Richard A. DeMillo, Nancy A. Lynch, Walter A. Burkhard, and Alfred V. Aho, editors, *Proceedings of the 11h Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1979, Atlanta, Georgia, USA*, pages 338–345. ACM, 1979. `doi:10.1145/800135.804426`.

12 Maxime Crochemore and Ely Porat. Fast computation of a longest increasing subsequence and application. *Inf. Comput.*, 208(9):1054–1059, 2010. `doi:10.1016/j.ic.2010.04.003`.

13 Omar Darwish and Amr Elmasry. Optimal time-space tradeoff for the 2d convex-hull problem. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 284–295. Springer, 2014. `doi: 10.1007/978-3-662-44777-2_24`.

14 Amr Elmasry, Torben Hagerup, and Frank Kammer. Space-efficient basic graph algorithms. In Ernst W. Mayr and Nicolas Ollinger, editors, *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany*, volume 30 of *LIPIcs*, pages 288–301. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. `doi:10.4230/LIPIcs.STACS.2015.288`.

15 Funda Ergün and Hossein Jowhari. On the monotonicity of a data stream. *Combinatorica*, 35(6):641–653, 2015. `doi:10.1007/s00493-014-3035-1`.

16 Greg N. Frederickson. Upper bounds for time-space trade-offs in sorting and selection. *J. Comput. Syst. Sci.*, 34(1):19–26, 1987. `doi:10.1016/0022-0000(87)90002-X`.

17 Michael L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975. `doi:10.1016/0012-365X(75)90103-X`.

18 Anna Gál and Parikshit Gopalan. Lower bounds on streaming algorithms for approximating the length of the longest increasing subsequence. *SIAM J. Comput.*, 39(8):3463–3479, 2010. `doi:10.1137/090770801`.

19 Parikshit Gopalan, T.S. Jayram, Robert Krauthgamer, and Ravi Kumar. Estimating the sortedness of a data stream. In *SODA 2007*, pages 318–327, 2007. URL: `http://dl.acm.org/citation.cfm?id=1283417`.

20 James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest subsequences. *Commun. ACM*, 20(5):350–353, 1977. `doi:10.1145/359581.359603`.

21 Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, 1997.

22 David Liben-Nowell, Erik Vee, and An Zhu. Finding longest increasing and common subsequences in streaming data. *J. Comb. Optim.*, 11(2):155–175, 2006. `doi:10.1007/s10878-006-7125-x`.

23 Andrea Lincoln, Virginia Vassilevska Williams, Joshua R. Wang, and R. Ryan Williams. Deterministic time-space trade-offs for k-sum. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 58:1–58:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. `doi:10.4230/LIPIcs.ICALP.2016.58`.

24 C. L. Mallows. Problem 62-2, patience sorting. *SIAM Review*, 4(2):143–149, 1962. URL: `http://www.jstor.org/stable/2028371`.

25 C. L. Mallows. Problem 62-2. *SIAM Review*, 5(4):375–376, 1963. URL: `http://www.jstor.org/stable/2028347`.

26 C. L. Mallows. Patience sorting. *Bulletin of the Institute of Mathematics and its Applications*, 9:216–224, 1973.

27 J. Ian Munro and Mike Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12:315–323, 1980. `doi:10.1016/0304-3975(80)90061-4`.

28 Timothy Naumovitz and Michael E. Saks. A polylogarithmic space deterministic streaming algorithm for approximating distance to monotonicity. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1252–1262. SIAM, 2015. `doi:10.1137/1.9781611973730.83`.

29   Noam Nisan. RL ⊆ SC. In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 619–623. ACM, 1992. `doi: 10.1145/129712.129772`.

30   Jakob Pagter and Theis Rauhe. Optimal time-space trade-offs for sorting. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, pages 264–268. IEEE Computer Society, 1998. `doi:10.1109/SFCS. 1998.743455`.

31   Michal Pilipczuk and Marcin Wrochna. On space efficiency of algorithms working on structural decompositions of graphs. In Nicolas Ollinger and Heribert Vollmer, editors, *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France*, volume 47 of *LIPIcs*, pages 57:1–57:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. `doi:10.4230/LIPIcs.STACS.2016.57`.

32   Prakash Ramanan. Tight $\Omega(n \lg n)$ lower bound for finding a longest increasing subsequence. *International Journal of Computer Mathematics*, 65(3–4):161–164, 1997. `doi:10.1080/ 00207169708804607`.

33   Dan Romik. *The surprising mathematics of longest increasing subsequences*. Cambridge University Press, 2015. `doi:10.1017/CBO9781139872003`.

34   Michael E. Saks and C. Seshadhri. Space efficient streaming algorithms for the distance to monotonicity and asymmetric edit distance. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1698–1709. SIAM, 2013. `doi: 10.1137/1.9781611973105.122`.

35   Michael E. Saks and C. Seshadhri. Estimating the longest increasing sequence in polylogarithmic time. *SIAM J. Comput.*, 46(2):774–823, 2017. `doi:10.1137/130942152`.

36   Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970. `doi:10.1016/S0022-0000(70)80006-X`.

37   Craige Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 13(2):179–191, 1961. `doi:10.4153/CJM-1961-015-3`.

38   Xiaoming Sun and David P. Woodruff. The communication and streaming complexity of computing the longest common and increasing subsequences. In *SODA 2007*, pages 336–345, 2007. URL: `http://dl.acm.org/citation.cfm?id=1283383.1283419`.

39   Joshua R. Wang. Space-efficient randomized algorithms for K-SUM. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014 - 22nd Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 810–829. Springer, 2014. `doi:10.1007/978-3-662-44777-2_67`.