Theory of Computing Systems (2020) 64:522–541
https://doi.org/10.1007/s00224-018-09908-6

Space-Efficient Algorithms for Longest Increasing Subsequence

Masashi Kiyomi[1] · Hirotaka Ono[2] · Yota Otachi[3] · Pascal Schweitzer[4] · Jun Tarui[5]

Published online: 22 January 2019

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 $O\left(\frac{1}{s} \cdot n^2 \cdot \log n\right)$ time for computing the length of a longest increasing subsequence, and $O\left(\frac{1}{s} \cdot n^2 \cdot \log^2 n\right)$ 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.

Keywords Longest increasing subsequence · Patience sorting · Space-efficient algorithm

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

This article is part of the Topical Collection on *Special Issue on Theoretical Aspects of Computer Science (2018)*

Partially supported by MEXT/JSPS KAKENHI grant numbers JP17H01698, JP18H04091, JP18K11168, JP18K11169, JP24106004. A preliminary version appeared in the proceedings of the 35th International Symposium on Theoretical Aspects of Computer Science (STACS 2018), vol. 96 of Leibniz International Proceedings in Informatics, pp. 44:1-44:15, 2018.

✉ Yota Otachi
otachi@cs.kumamoto-u.ac.jp

Extended author information available on the last page of the article.

that uses $O(n \log n)$ bits of working space [2, 17, 37], where n is the length of the sequence.

A wide-spread algorithm achieving these bounds is Patience Sorting, devised by Mallows [24–26]. 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 $O\left(\log^2 n\right)$ 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 $O\left(\sqrt{n} \log n\right)$ bits which runs in $O\left(n^{1.5} \log n\right)$ 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 $O\left(\frac{1}{s} \cdot n^2 \log n\right)$ 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, $O\left(\sqrt{n} \log n\right)$ bits and runs in $O\left(n^{1.5} \log n\right)$ 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 $O\left(\frac{1}{s} \cdot n^3 \log n\right)$. However, we show that one can do much better, achieving a running time of $O\left(\frac{1}{s} \cdot n^2 \log^2 n\right)$ 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 [2, 17, 37], 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 [6, 12, 20]. Patience Sorting, an efficient algorithm for LIS, has been a research topic in itself, especially in the context of Young tableaux [2, 8, 24–26, 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 [15, 18, 19, 28, 34, 35, 38]. 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 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 [7, 16, 27, 30], graph searching [4, 9, 14, 31], geometric computation [1, 5, 10, 13], and k-SUM [23, 39].

2 Preliminaries

Let $\tau = \langle \tau(1), \tau(2), \dots, \tau(n) \rangle$ be a sequence of n integers, possibly with repetitions. For $1 \le i_1 < \cdots < i_\ell \le n$, we denote by $\tau[i_1, \dots, i_\ell]$ the *subsequence* $\langle \tau(i_1), \dots, \tau(i_\ell) \rangle$ of τ. A subsequence $\tau[i_1, \dots, i_\ell]$ is an *increasing subsequence* of τ if $\tau(i_1) < \cdots < \tau(i_\ell)$. If $\tau(i_1) \le \cdots \le \tau(i_\ell)$, 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.

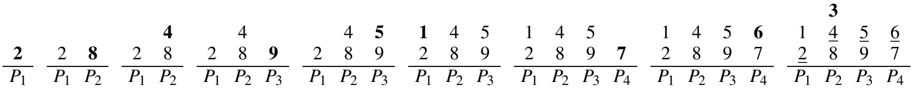
Internally, the algorithm maintains a collection of piles. A *pile* is a stack of integers. It is equipped with the procedures push and top: the push procedure appends a new element to become the new top of the pile; and the top procedure simply returns the element on top of the pile, which is always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

We describe how Patience Sorting computes $\mathsf{lis}(\tau)$. 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 $\mathsf{lis}(\tau)$ (Line 8).

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 $\tau(i) > \mathtt{top}(P_\ell)$ then
4:         increment $\ell$, let $P_\ell$ be a new empty pile, and set $j := \ell$
5:     else
6:         set $j$ to be the smallest index with $\tau(i) \le \mathtt{top}(P_j)$
7:     push $\tau(i)$ to $P_j$
8: return $\ell$
```

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 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.



Proposition 3.1 ([2, 17, 37]) *Given a sequence τ of length n,* Patience Sorting *computes* $\mathsf{lis}(\tau)$ *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 $\mathsf{lis}(\tau) \leq \ell$.

Now we show that $\mathsf{lis}(\tau) \geq \ell$. 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.

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, $\texttt{top}(P_k) < \texttt{top}(P_{k'})$ 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 $1 \leq k < k' \leq \ell$, we have $\texttt{top}(P_k) < \texttt{top}(P_{k'})$ if P_k and $P_{k'}$ are nonempty.

Proof We prove the statement by contradiction. Let i be the first index for which Patience Sorting pushes $\tau(i)$ to some pile P_j, so that the statement of the observation becomes false.

First assume that $\mathtt{top}(P_j) \geq \mathtt{top}(P_{j'})$ for some $j' > j$. Let $\tau(i')$ be the element in P_j pushed to the pile right before $\tau(i)$. By the definition of Patience Sorting, it holds that

$$\tau(i') \geq \tau(i) = \mathtt{top}(P_j) \geq \mathtt{top}(P_{j'}).$$

This contradicts the minimality of i because $\tau(i')$ was the top element of P_j before $\tau(i)$ was pushed to P_j.

Next assume that $\mathtt{top}(P_{j'}) \geq \mathtt{top}(P_j)$ for some $j' < j$. This case contradicts the definition of Patience Sorting since $\tau(i) = \mathtt{top}(P_j) \leq \mathtt{top}(P_{j'})$ and thus $\tau(i)$ actually has to be pushed to a pile with an index smaller or equal to j'. □

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 $O(n)$. By storing its position in the sequence, each element in the piles can be represented with $O(\log n)$ bits (instead of $O(\log |\Sigma|)$ bits, where Σ is the alphabet from which the input elements are taken). Thus Patience Sorting consumes $O(n \log n)$ bits in total. 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 for Permutations

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 element of $\{1, \ldots, n\}$ assuming that each element appears exactly once in the sequence. Hence, this algorithm cannot be applied to general inputs where repetitions may happen and the alphabet size $|\Sigma|$ can be much larger than the input length n.

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 $\{i_1, \ldots, i_\ell\} = \{i : \tau(i) = \min\{\tau(j) : 1 \leq j \leq i\}\}$. 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 below 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.

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

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

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 $O\left(\frac{1}{s} \cdot n^2 \log n\right)$ 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 $i + 1 \le k \le \min\{j, \mathsf{lis}(\tau)\}$ 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 appropraiate 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 $h \le j$.
- 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 $h \le j$.

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 $i < j \le \mathsf{lis}(\tau)$, 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 $\sqrt{n} \le s \le n$ and a sequence τ of length n, computes $\mathsf{lis}(\tau)$ in $O\left(\frac{1}{s} \cdot n^2 \log n\right)$ time with $O(s \log n)$ bits of space.*

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

```
set $r := 0$                              ▷ $r$ points to the most recently read element in $P_i$
set $\ell := i$                           ▷ the largest index of the piles constructed so far
initialize $p_{i+1}, \ldots, p_j$ to $\infty$        ▷ $p_k$ is the element currently on top of $P_k$
initialize $c_{i+1}, \ldots, c_j$ to 0               ▷ $c_k$ is the current size of $P_k$
for $x = 1$ to $n$ do
    ▷ filtering out irrelevant elements
    if $\tau(x)$ is $P_i(r + 1)$ then
        increment $r$ and continue the for-loop
    else if $\tau(x) < P_i(r)$ or ($\ell \ge j$ and $\tau(x) > p_j$) then
        ignore the element and continue the for-loop
    ▷ push $\tau(x)$ to the appropriate pile
    if $\tau(x) > p_\ell$ then
        increment $\ell$ and set $h := \ell$
    else
        set $h$ to be the smallest index with $\tau(i) < p_h$
    set $p_h := \tau(x)$ and increment $c_h$
```

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 $i \ge 0$ we computed the pile P_i of size at most s explicitly. We repeat the following process until we find $\mathsf{lis}(\tau)$.

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

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

In each iteration, we first compute the size $|P_k|$ for $i+1 \le k \le i+2s$. 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 $i + s + 1 \le j \le i + 2s$ and $|P_j| \le n/s$. Since $s \ge \sqrt{n}$, it holds that $|P_j| \le n/\sqrt{n} = \sqrt{n} \le s$. 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 a working space of $O(s \log n)$ 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 $O\left(\frac{1}{s} \cdot n^2 \log n\right)$. □

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

Corollary 4.5 *Given a sequence τ of length n, $\mathsf{lis}(\tau)$ can be computed in $O\left(n^{1.5} \log n\right)$ time with $O\left(\sqrt{n} \log n\right)$ 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 $O\left(\frac{1}{s} \cdot n^3 \log n\right)$.

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 $O\left(n^{1.5} \log^2 n\right)$ time with $O\left(\sqrt{n} \log n\right)$ 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 $O\left(\frac{1}{s} \cdot n^2 \log n\right)$ time with $O(s \log n)$ bits by running the algorithm from the previous section twice, once in the ordinary

[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.

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 $O\left(\frac{1}{s} \cdot n^2 \log n\right)$. To remember the path to the current recursion, we need some additional space, but it is bounded by $O\left(\log^2 n\right)$ bits.

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 $O\left(\frac{1}{s} \cdot n^2 \log n\right)$ 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

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

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.

Algorithm 5 Reverse patience sorting.

```
set $\ell := 0$ and initialize the dummy pile $Q_0$ with the single element $+\infty$
for $i = n$ to 1 do
    if $\tau(i) < \mathtt{top}(Q_\ell)$ then
        increment $\ell$, let $Q_\ell$ to be a new empty pile, and set $j := \ell$
    else
        set $j$ to be the smallest index with $\tau(i) > \mathtt{top}(Q_j)$
    push $\tau(i)$ to $Q_j$
return $\ell$
```

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.

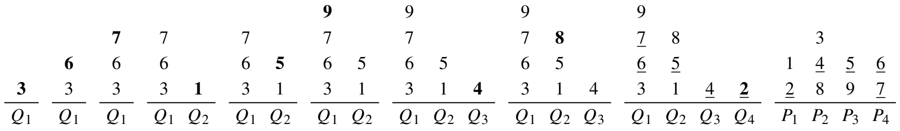
Observation 5.5 $\tau(i) \in Q_j$ if and only if a longest increasing subsequence of τ starting at $\tau(i)$ has length j.

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 $i_\ell - k + 1 = \mathsf{lis}(\tau) - k + 1$, it holds that $\tau(k) \in Q_{\mathsf{lis}(\tau)-k+1}$. □

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 underlined.



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 $O\left(\frac{1}{s} \cdot n^2 \log n\right)$ 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 $\tau[1, \dots, j-1]$ formed by the elements smaller than $\tau(j)$. Similarly let τ_R be the subsequence of $\tau[j+1, \dots, |\tau|]$ 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.*

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

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 $\mathsf{lis}(\tau')$, 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.

Algorithm 6 Recursively finding a longest increasing subsequence of ρ.

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

5.5 Time and Space Complexity

In Theorem 5.1, the claimed running time is $O\left(\frac{1}{s} \cdot n^2 \log^2 n\right)$. 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 $O\left(\frac{1}{s} \cdot n^2 \log n\right)$. 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* $\log_{6/5} \mathit{lis}(\tau')$*, where τ' is the subsequence of τ computed in Line 3.*

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

Since $\ell > 3|\tau'|/s$, Algorithm 6 recurses 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* $O\left(\frac{1}{s} n^2 \log n\right)$ *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 $O\left(\frac{1}{s}|\tau'|^2 \log |\tau'|\right)$. Thus the total running time at a depth is $O\left(\sum_{\tau'} \frac{1}{s}|\tau'|^2 \log |\tau'|\right)$, which is $O\left(\frac{1}{s}n^2 \log n\right)$ 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.

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 high communication complexity [19, 22, 38].

Proposition 6.1 ([19, 38]) *Let π be a permutation of $\{1, \ldots, 2n\}$. Given the first half of π to Alice and the second half to Bob, they need $\Omega(n)$ bits of communication to compute $\mathsf{lis}(\pi)$ in the worst case (even with 2-sided error randomization).*

Now we present our lower bound. Note that the lower bound even holds for the special case where input is restricted to permutations.

Theorem 6.2 *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.*

Proof Given an arbitrary $n > 1$, let π' be a permutation of $\{1, \ldots, 2n\}$. We construct a permutation π of $\{1, \ldots, 4n\}$ as follows. Let $\pi'_1 = \langle \pi(1), \ldots, \pi(n) \rangle$ be the first half of π', define $\pi'_2 = \langle 4n, 4n-1, \ldots, 2n+2 \rangle$ and let $\pi'_3 = \langle \pi(n+1), \pi(n+2), \ldots, \pi(2n) \rangle$ be the second half of π'. Then we define π to be the concatenation of π'_1, π'_2, π'_3 and the one element sequence $\pi'_4 = \langle 2n+1 \rangle$, in that order.

It is not difficult to see that π is a permutation and that $\mathsf{lis}(\pi) = \mathsf{lis}(\pi') + 1$. To see the latter, observe that the concatenation of π'_2 and π'_4 is a decreasing subsequence of π. Hence any increasing subsequence of π can contain at most one element not in π'. On the other hand, any increasing subsequence of π' of length ℓ can be extended with the element $2n+1$ of π'_4 to an increasing subsequence of π of length $\ell + 1$.

We say a sequential access algorithm traverses the middle if it accesses a position in π'_1 and then accesses a position in π'_3 or vice versa with possibly accessing elements in π'_2 but only such elements in meantime. Since each traversal of the middle takes $\Omega(n)$ time, it suffices to show that the number of traversals of the middle is $\Omega(n/b)$.

Suppose we are given a sequential access algorithm M that computes $\mathsf{lis}(\pi)$ with t traversals of the middle. Using M, we construct a two-player communication protocol for computing $\mathsf{lis}(\pi')$ with at most tb bits of communication. (A similar technique is described for streaming algorithms in [38]).

Recall that the first player Alice gets the first half π'_1 of π' and the second player Bob gets the second half π'_3 of π'. They compute $\mathsf{lis}(\pi')$ together as follows.

- Before starting computation, Alice computes π_A by concatenating π'_1 and π'_2 in that order, and Bob computes π_B by concatenating π'_2, π'_3, and π'_4 in that order.
- They first compute $\mathsf{lis}(\pi)$ using M by repeating the following phases:
    - – Alice starts the computation by M and continues while M stays in $\pi[1, \ldots, 3n-1] = \pi_A$. When M tries to access $\pi[3n, \ldots, 4n]$, and thus a traversal of the middle occurs, Alice stops and sends all b bits stored by M to Bob.
    - – Bob restores the b bits received from Alice to the working memory of M and continues computation while M stays in $\pi[n+1, \ldots, 4n] = \pi_B$. A traversal of the middle is occurred when M tries to access $\pi[1, \ldots, n]$. Bob then stops and sends the b bits currently stored by M back to Alice.
- When M outputs $\mathsf{lis}(\pi)$ and terminates, the currently active player outputs $\mathsf{lis}(\pi) - 1$ as $\mathsf{lis}(\pi')$ and terminates the protocol.

The two players correctly simulate M and, as a result, compute $\mathsf{lis}(\pi')$ together. Since the algorithm M invokes t traversals, the total number of bits sent is at most tb. Since $tb \in \Omega(n)$ holds by Proposition 6.1, we have $t \in \Omega(n/b)$ as required. □

Recall that our algorithms for the LIS problem use $O(s \log n)$ bits and run in $O(\frac{1}{s}n^2 \log n)$ time for computing the length and in $O(\frac{1}{s}n^2 \log^2 n)$ time for finding a subsequence, where $\sqrt{n} \le s \le n$. By Theorem 6.2, their time complexity is optimal for algorithms with sequential access up to polylogarithmic factors of $\log^2 n$ and $\log^3 n$, respectively.

7 Concluding Remarks

Our results in this paper raise the following question:

Can we solve LIS with $o\left(\sqrt{n}\right)$-space in polynomial time?

Observe that an unconditional 'no' implies that SC $\neq$ P $\cap$ 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]. As a conditional lower bound, we can use the (hypothetical) hardness of LONGEST COMMON SUBSEQUENCE (LCS). Given two strings, LCS asks to find a longest sequence that is a subsequence of both strings. It is easy to see that a longest increasing subsequence of a sequence τ can be computed as a longest common subsequence of τ and the sequence obtained by sorting τ. The other direction is not that obvious, but there is a reduction from LCS on strings of length n to LIS on a string of length at most n^2 [20]. The reduction can be easily implemented in log-space. This implies that an $O(f(n))$-space polynomial-time algorithm for LIS can be used as an $O(f(n^2))$-space polynomial-time algorithm for LCS. In particular, an $O(\sqrt{n}^{\,1-\epsilon})$-space polynomial-time algorithm for LIS gives an $O(n^{1-\epsilon})$-space polynomial-time algorithm for LCS for any $\epsilon > 0$, and a log-space algorithm for LIS implies a log-space algorithm for LCS.

To make the presentation simple, we used the length n of τ to bound $\mathsf{lis}(\tau)$ in the time complexity analyses of the algorithms. If we analyze the complexity in terms of $\mathsf{lis}(\tau)$ instead of n when possible, we can obtain the following *output-sensitive* bounds.

Theorem 7.1 *Let s be an integer satisfying $\sqrt{n} \le s \le n$, and let τ be a sequence of length n with $\mathsf{lis}(\tau) = k$. Using $O(s \log n)$ bits of space, $\mathsf{lis}(\tau)$ can be computed in $O\left(\frac{1}{s} \cdot kn \log k\right)$ time and a longest increasing subsequence of τ can be found in $O\left(\frac{1}{s} \cdot kn \log^2 k\right)$ time.*

Acknowledgments The authors are grateful to William S. Evans for bringing the reduction from LCS to LIS mentioned in the concluding remarks to their attention.

References

1. Ahn, H.-K., Baraldo, N., Oh, E., Silvestri, F.: A time-space trade-off for triangulations of points in the plane. In: COCOON 2017, pp. 3–12 (2017). https://doi.org/10.1007/978-3-319-62389-4_1
2. Aldous, D., Diaconis, P.: Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem. Bull. Am. Math. Soc. **36**(4), 413–432 (1999). https://doi.org/10.1090/S0273-0979-99-00796-X
3. Asano, T., Elmasry, A., Katajainen, J.: Priority queues and sorting for read-only data. In: TAMC 2013, pp. 32–41 (2013). https://doi.org/10.1007/978-3-642-38236-9_4
4. Asano, T., Izumi, T., Kiyomi, M., Konagaya, M., Ono, H., Otachi, Y., Schweitzer, P., Tarui, J., Uehara, R.: Depth-first search using $O(n)$ bits. In: ISAAC 2014, pp. 553–564 (2014). https://doi.org/10.1007/978-3-319-13075-0_44
5. Banyassady, B., Korman, M., Mulzer, W., Van Renssen, A.ndré., Roeloffzen, M., Seiferth, P., Stein, Y.: Improved time-space trade-offs for computing Voronoi diagrams. In: STACS 2017, vol. 66, pp. 9:1–9:14 (2017). https://doi.org/10.4230/LIPIcs.STACS.2017.9
6. Bespamyatnikh, S., Segal, M.: Enumerating longest increasing subsequences and patience sorting. Inf. Process. Lett. **76**(1–2), 7–11 (2000). https://doi.org/10.1016/S0020-0190(00)00124-1
7. Borodin, A., Cook, S.: A time-space tradeoff for sorting on a general sequential model of computation. SIAM J. Comput. **11**(2), 287–297 (1982). https://doi.org/10.1137/0211022
8. Burstein, A., Lankham, I.: Combinatorics of patience sorting piles. Séminaire Lotharingien de Combinatoire **54A**, B54Ab (2006). http://www.mat.univie.ac.at/~slc/wpapers/s54Aburlank.html
9. Chakraborty, S., Satti, S.R.: Space-efficient algorithms for maximum cardinality search, stack BFS, queue BFS and applications. In: COCOON 2017, pp. 87–98 (2017). https://doi.org/10.1007/978-3-319-62389-4_8
10. Chan, T.M., Chen, E.Y.: Multi-pass geometric algorithms. Discret. Comput. Geom. **37**(1), 79–102 (2007). https://doi.org/10.1007/s00454-006-1275-6
11. Cook, S.A.: Deterministic CFL's are accepted simultaneously in polynomial time and log squared space. In: STOC 1979, pp. 338–345 (1979). https://doi.org/10.1145/800135.804426
12. Crochemore, M., Porat, E.: Fast computation of a longest increasing subsequence and application. Inf. Comput. **208**(9), 1054–1059 (2010). https://doi.org/10.1016/j.ic.2010.04.003
13. Darwish, O., Elmasry, A.: Optimal time-space tradeoff for the 2D convex-hull problem. In: ESA 2014, pp. 284–295 (2014). https://doi.org/10.1007/978-3-662-44777-2_24
14. Elmasry, A., Hagerup, T., Kammer, F.: Space-efficient basic graph algorithms. In: STACS 2015, vol. 30, pp. 288–301 (2015). https://doi.org/10.4230/LIPIcs.STACS.2015.288
15. Ergun, F., Jowhari, H.: On the monotonicity of a data stream. Combinatorica **35**(6), 641–653 (2015). https://doi.org/10.1007/s00493-014-3035-1
16. Frederickson, G.N.: Upper bounds for time-space trade-offs in sorting and selection. J Comput Syst Sci **34**(1), 19–26 (1987). https://doi.org/10.1016/0022-0000(87)90002-X
17. Fredman, M.L.: On computing the length of longest increasing subsequences. Discret. Math. **11**(1), 29–35 (1975). https://doi.org/10.1016/0012-365X(75)90103-X
18. Gál, A., Gopalan, P.: Lower bounds on streaming algorithms for approximating the length of the longest increasing subsequence. SIAM J. Comput. **39**(8), 3463–3479 (2010). https://doi.org/10.1137/090770801
19. Gopalan, P., Jayram, T.S., Krauthgamer, R., Kumar, R.: Estimating the sortedness of a data stream. In: SODA 2007, pp. 318–327 (2007). http://dl.acm.org/citation.cfm?id=1283417
20. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest common subsequences. Commun. ACM **20**(5), 350–353 (1977). https://doi.org/10.1145/359581.359603
21. Kushilevitz, E., Nisan, N.: Communication complexity. Cambridge University Press, Cambridge (1997)
22. Liben-Nowell, D., Vee, E., An, Z.: Finding longest increasing and common subsequences in streaming data. J. Comb. Optim. **11**(2), 155–175 (2006). https://doi.org/10.1007/s10878-006-7125-x
23. Lincoln, A., Williams, V.V., Wang, J.R., Ryan Williams, R.: Deterministic time-space trade-offs for k-SUM. In: ICALP 2016, pp. 58:1–58:14 (2016). https://doi.org/10.4230/LIPIcs.ICALP.2016.58
24. Mallows, C.L.: Problem 62-2, patience sorting. SIAM Rev. **4**(2), 143–149 (1962). http://www.jstor.org/stable/2028371
25. Mallows, C.L.: Problem 62-2. SIAM Rev. **5**(4), 375–376 (1963). http://www.jstor.org/stable/2028347

26. Mallows, C.L.: Patience sorting. Bulletin of the Institute of Mathematics and its Applications **9**, 216–224 (1973)
27. Munro, J.I., Paterson, M.S.: Selection and sorting with limited storage. Theor. Comput. Sci. **12**(3), 315–323 (1980). https://doi.org/10.1016/0304-3975(80)90061-4
28. Naumovitz, T., Saks, M.: A polylogarithmic space deterministic streaming algorithm for approximating distance to monotonicity. In: SODA 2015, pp. 1252–1262 (2015). https://doi.org/10.1137/1.9781611973730.83
29. Nisan, N.: RL $\subseteq$ SC. In: STOC 1992, pp. 619–623 (1992). https://doi.org/10.1145/129712.129772
30. Pagter, J., Rauhe, T.: Optimal time-space trade-offs for sorting. In: FOCS 1998, pp. 264–268 (1998). https://doi.org/10.1109/SFCS.1998.743455
31. Pilipczuk, M., Wrochna, M.: On space efficiency of algorithms working on structural decompositions of graphs. In: STACS 2016, vol. 47, pp. 57:1–57:15 (2016). https://doi.org/10.4230/LIPIcs.STACS.2016.57
32. Ramanan, P.: Tight $\Omega(n \lg n)$ lower bound for finding a longest increasing subsequence. Int. J. Comput. Math. **65**(3–4), 161–164 (1997). https://doi.org/10.1080/00207169708804607
33. Romik, D.: The surprising mathematics of longest increasing subsequences. Cambridge University Press, Cambridge (2015). https://doi.org/10.1017/CBO9781139872003
34. Saks, M., Seshadhri, C.: Space efficient streaming algorithms for the distance to monotonicity and asymmetric edit distance. In: SODA 2013, pp. 1698–1709 (2013). https://doi.org/10.1137/1.9781611973105.122
35. Saks, M., Seshadhri, C.: Estimating the longest increasing sequence in polylogarithmic time. SIAM J. Comput. **46**(2), 774–823 (2017). https://doi.org/10.1137/130942152
36. Savitch, W.J.: Relationships between nondeterministic and deterministic tape complexities. J. Comput. Syst. Sci. **4**(2), 177–192 (1970). https://doi.org/10.1016/S0022-0000(70)80006-X
37. Schensted, C.: Longest increasing and decreasing subsequences. Can. J. Math. **13**(2), 179–191 (1961). https://doi.org/10.4153/CJM-1961-015-3
38. Su, X., Woodruff, D.P.: The communication and streaming complexity of computing the longest common and increasing subsequences. In: SODA 2007, pp. 336–345 (2007). http://dl.acm.org/citation.cfm?id=1283383.1283419
39. Wang, J.R.: Space-efficient randomized algorithms for K-SUM. In: ESA 2014, pp. 810–829 (2014). https://doi.org/10.1007/978-3-662-44777-2_67

Affiliations

Masashi Kiyomi[1] · Hirotaka Ono[2] · Yota Otachi[3] · Pascal Schweitzer[4] · Jun Tarui[5]

Masashi Kiyomi
masashi@yokohama-cu.ac.jp

Hirotaka Ono
ono@nagoya-u.jp

Pascal Schweitzer
schweitzer@cs.uni-kl.de

Jun Tarui
tarui@ice.uec.ac.jp

[1] Yokohama City University, Yokohama, Japan

[2] Nagoya University, Nagoya, Japan

[3] Kumamoto University, Kumamoto, Japan

[4] TU Kaiserslautern, Kaiserslautern, Germany

[5] The University of Electro-Communications, Chofu, Japan