Space-Efficient Algorithms for Longest Increasing Subsequence (Extended Abstract)

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

Abstract: Let π be a permutation of $\{1,\dots,n\}$. For $1 \le i_1 < \cdots < i_\ell \le n$, a *subsequence* $\pi\langle i_1,\dots,i_\ell\rangle$ of π is the sequence $\langle\pi(i_1),\dots,\pi(i_\ell)\rangle$. A subsequence $\pi\langle i_1,\dots,i_\ell\rangle$ is an *increasing subsequence* of π if $\pi(i_1) < \cdots < \pi(i_\ell)$. In this extended abstract, we present space-efficient algorithms for finding a longest increasing subsequence of a given permutation. Using $\sqrt{n}$ words of additional working space, our algorithm solves the problem in $O(n^{1.5}\log^2 n)$ time. It runs in $O(n^{1.5}\log n)$ time when we need only the length of a longest increasing subsequence.

1. Introduction

Given a sequence of ordered (totally or partially) elements, the problem of finding a longest increasing subsequence is studied in many settings (see [8] and the references therein) and known to have an $O(n\log n)$-time deterministic exact algorithm that uses additional space of $O(n\log n)$ bits [1], [4], [10], where n is the length of the sequence.

In this paper, we present the first space-efficient deterministic exact algorithm for the longest increasing subsequence problem (LIS). We say that an algorithm for LIS is *space-efficient* if it uses $o(n)$ bits and runs in polynomial time. We ask for $o(n)$ bits since an $O(n^2)$-time algorithm with $O(n)$ bits can be obtained by slightly modifying a known algorithm. Also, we can observe that a Savitch type algorithm [9] for this problem uses $O(\log^2 n)$ bits and runs in quasipolynomial time. We deeply study the internal states of the known algorithm and show that by only keeping some part of a state we can simulate it. For any $\sqrt{n} \le s \le n$, our algorithm for computing the length of a longest increasing subsequence uses $O(s\log n)$ bits and runs in $O(\frac{1}{s}n^2\log n)$ time. When $s = n$ our algorithm is equivalent to the known algorithm, and when $s = \sqrt{n}$ it uses $O(\sqrt{n}\log n)$ bits and runs in $O(n^{1.5}\log n)$ time. To find an actual subsequence, one can run our algorithm n times since in each run one can find the last element in a longest increasing subsequence. We show that we can do much better and present an $O(\frac{1}{s}n^2\log^2 n)$-time algorithm. Finally we show that the running time of our algorithm (for the length) is best possible as a comparison-based multi-pass algorithm. An algorithm is a *multi-pass algorithm* if it can read the input only from left to right and can repeat it multiple times.

For the sake of simplicity, we assume in the rest of the paper that the input is a permutation of $\{1,\dots,n\}$ rather than a general sequence over an ordered alphabet. Since our algorithms are comparison based, we can easily modify the algorithms to work for general sequences with possibly repeated entries. There are $O(n\log\log n)$-time algorithms known for permutations [2], [6]. However, those algorithms are not comparison-based and thus cannot be applied to the general case. It is known that any comparison-based algorithm needs $\Omega(n\log n)$ comparisons even for only computing the length of a longest increasing subsequence [4], [7].

In this extended abstract, most of the proofs are omitted. A full version will be available online soon.

2. Preliminaries

Let π be a permutation of $\{1,\dots,n\}$. For $1 \le i_1 < \cdots < i_\ell \le n$, a *subsequence* $\pi\langle i_1,\dots,i_\ell\rangle$ of π is the sequence $\langle\pi(i_1),\dots,\pi(i_\ell)\rangle$. A subsequence $\pi\langle i_1,\dots,i_\ell\rangle$ is an *increasing subsequence* of π if $\pi(i_1) < \cdots < \pi(i_\ell)$. A *decreasing subsequence* is defined analogously. By $\mathsf{lis}(\pi)$, we denote the length of a longest increasing subsequence of π.

For example, consider a permutation $\pi_1 = 284951763$[*1] of $\{1,\dots,9\}$. It has an increasing subsequence $\pi_1\langle 1,3,5,8\rangle = \langle 2,4,5,6\rangle$. Since there is no increasing subsequence of π_1 with length 5 or more, we have $\mathsf{lis}(\pi_1) = 4$.

In the computational model in this paper, 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. In the best case, we hope to have a time/space tread-off. For example, if we have a traditional algorithm of running time $f(n)$, then we want to have an

[1] Yokohama City University, Yokohama, Japan.
masashi@yokohama-cu.ac.jp
[2] Nagoya University, Nagoya, Japan.
ono@i.nagoya-u.ac.jp
[3] Kumamoto University, Kumamoto, Japan.
otachi@cs.kumamoto-u.ac.jp
[4] RWTH Aachen University, Aachen, Germany.
schweitzer@informatik.rwth-aachen.de
[5] The University of Electro-Communications, Tokyo, Japan.
tarui@ice.uec.ac.jp

[*1] By this sloppy notation, we mean $\pi_1(1) = 2$, $\pi_1(2) = 8$, $\pi_1(3) = 4$, and so on.

$f(n)/g(s)$-time and s-space algorithm for some (good) function g.

3. Patience Sorting

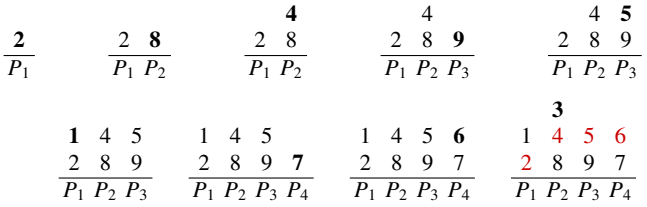
Since all our algorithms are based on the classical algorithm Patience Sorting, here we describe it in detail and recall some important properties of its internal states.

Given a permutation π of $\{1,\dots,n\}$, Patience Sorting computes $\mathsf{lis}(\pi)$ in $O(n\log n)$ time using $O(n\log n)$ bits working space. See Algorithm 1.

Algorithm 1 Patience Sorting

```
1: set ℓ := 0 and initialize the dummy pile P_0 with the single element −∞
2: for 1 ≤ i ≤ n do
3:     if π(i) > top(P_ℓ) then
4:         increment ℓ, let P_ℓ to be a new empty pile, and set j := ℓ
5:     else
6:         set j to be the smallest index with π(i) ≤ top(P_j)
7:     push π(i) to P_j.
8: return ℓ
```

Now use the permutation $\pi_1 = 284951763$ again for an example. The following shows the execution of Algorithm 1 on π_1. The red elements in the final piles form a longest increasing subsequence $\pi_1\langle 1,3,5,8\rangle = \langle 2,4,5,6\rangle$, which can be obtained as described above.



3.1 Time and space complexity

To see that Patience Sorting runs in $O(n\log n)$ time, observe that at any point of the execution, the top elements of the piles are ordered increasingly from left to right (see [3]). Namely, $\mathtt{top}(P_k) \le \mathtt{top}(P_{k'})$ for $1 \le k < k' \le \ell$. Thus the 6th step in Algorithm 1 can be done in $O(\log n)$ time by binary search. Note that to compute $\mathsf{lis}(\pi)$, it suffices to remember the top elements of the piles. However, the algorithm still uses $\Omega(n\log n)$ bits when $\mathsf{lis}(\pi) \in \Omega(n)$.

3.2 Correctness

First observe that each pile is a decreasing subsection of π. Thus, ℓ is an upper bound of $\mathsf{lis}(\pi)$.

If we keep all elements in the piles, we can compute a longest increasing subsequence, in the reversed order, as follows [1]:

(1) Pick any element of P_ℓ;

(2) For $1 \le i < \ell$, let $\pi(h)$ be the element picked from P_{i+1}. Pick the element $\pi(h')$ that was on the top of P_i when $\pi(h)$ had been pushed to P_{i+1}. Clearly, $h' < h$ and $\pi(h') < \pi(h')$.

3.3 An $O(n)$-space $O(n^2)$-time algorithm

Here we observe that $\mathsf{lis}(\pi)$ can be computed with $O(n)$ bits of additional space. The following facts guarantee the correctness of Algorithm 2.

Proposition 3.1 ([3]). *For each element $\pi(i)$ in P_j, the longest increasing subsequence ending at $\pi(i)$ has length j.*

Proposition 3.2 ([3]). *If the elements included in $P_1,\dots,P_k$ are known, the next pile P_{k+1} can be constructed by greedily taking unused elements to form a maximal decreasing subsequence.*

Algorithm 2 Computing $\mathsf{lis}(\pi)$ with $O(n)$ bits

```
1: set ℓ := 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 := ∞.
4:     for 1 ≤ i ≤ n do                ▷ this for-loop constructs P_ℓ implicitly
5:         if π(i) is unused and π(i) < t then
6:             mark π(i) as "used" and set t := π(i).   ▷ t is the current top of P_ℓ
7: return ℓ
```

Theorem 3.3. *For a permutation π of $\{1,\dots,n\}$, $\mathsf{lis}(\pi)$ can be computed in $O(n^2)$ time using $n + O(\log n)$ bits working space.*

4. Algorithm for computing the length

In this section, we present our main algorithm that computes $\mathsf{lis}(\pi)$ with $O(s\log n)$ bits and $O(\frac{1}{s}n^2\log n)$ time for any $\sqrt{n} \le s \le n$.

Observation 4.1. *Let x be an element of π not included in P_i, and let y be the smallest element of P_i positioned before x. Then, x is to the left of P_i if and only if $x < y$.*

Proof. When the algorithm accessed x in the permutation π, the top element of P_i was y. If $x > y$, then the algorithm pushed x to a pile P_j with $j > i$. Assume $x < y$. In this case, the algorithm pushed x to a pile P_j for some $h \le i$. Since the algorithm did not push x into P_i, x is to the left of P_i. □

Lemma 4.2. *Given P_i and an index $j > i$, the size $|P_k|$ for all $i+1 \le k \le j$ can be computed in $O(n\log n)$ time with $O(|P_i|+j-i)$ words.*

Proof. Recall that Algorithm 1 (Patience Sorting) scans the permutation π from left to right, and put each element to the correct pile. We still do the same except that we ignore the elements to the left of P_i and the ones to the right of P_j.

To ignore the elements to the left of P_i, we maintain the index j of P_i that points to the element read most recently in the scan. When we read a new element x, we have three cases.

- If $|P_i| \ge j+1$ and $x = P_i(j+1)$, then we just update (increment) the index j and read the next element.
- Else if $x < P_i(j)$, then x is ignored as it is to the left of P_i.
- Otherwise, we have $x \ge P_i(j)$. In this case, x is normally processed.

Ignoring the elements to the right of P_j is easier. Let x be the newly read element.

- If no part of P_j is constructed, then x is normally processed.
- Otherwise, we compare x and the current top element y of P_j.
  - If $x > y$, then x is to the right of P_j, and thus ignored.
  - Otherwise x is normally processed.

By ignoring irrelevant elements as described above, we can run Patience Sorting only for the piles we need. We only keep the top elements of the piles and additionally store the size of each pile. Thus the additional space is as we claimed in the statement of the lemma. The running times is the same as the one of Patience Sorting since we only need constant number of additional steps for each step to ignoring irrelevant elements. □

Lemma 4.3. *Given P_i and an index $j > i$, we can compute P_j in $O(n \log n)$ time with $O(|P_i| + |P_j| + j - i)$ words.*

Proof. The proof is almost equivalent to the one of Lemma 4.2. We just need to keep all the elements of P_j additionally. □

Now we are ready to prove our first main result.

Theorem 4.4. *Let s be an integer satisfying $\sqrt{n} \le s \le n$. Given a permutation π of $\{1, \ldots, n\}$, the length of a longest increasing subsequence, $\mathsf{lis}(\pi)$, can be computed in $O(\frac{1}{s} n^2 \log n)$ time using $O(s \log n)$ bits of additional space.*

Proof. To apply Lemma 4.2 for the first step, we start with a dummy pile P_0 with a dummy entry $\pi(0) = -\infty$. In the following, assume that for some $i \ge 0$ we have the pile P_i of size at most s.

We first compute the size $|P_k|$ for $i + 1 \le k \le i + 2s$. During this process, we may find $\mathsf{lis}(\pi) \le i + 2s$. In such a case we output $\mathsf{lis}(\pi)$ 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 and replace i with j. We repeat this process until we find $\mathsf{lis}(\pi)$.

Clearly, the additional space used is $O(s \log n)$ bits. Each pass runs in $O(n \log n)$ time. There are at most $\mathsf{lis}(\pi)/s$ passes since each pass makes at least s steps of progress. Since $\mathsf{lis}(\pi) \le n$, the total running time is $O(\frac{1}{s} n^2 \log n)$. □

See Algorithm 3 for a pseudocode of the algorithm described in the proof of Theorem 4.4.

Algorithm 3 Computing $\mathsf{lis}(\pi)$ with $O(s \log n)$ bits

```
1: set ℓ := 0 and initialize the dummy pile P_0 with the single element −∞
2: for ever do
3:     compute the size of P_i for all ℓ + 1 ≤ i ≤ ℓ + 2s
4:     if we find lis(π) ≤ ℓ + 2s then
5:         return lis(π)
6:     let j be the largest index |P_j| ≤ s and ℓ + 1 ≤ j ≤ ℓ + 2s
                                                  ▷ j ≥ ℓ + s + 1
7:     compute P_j and set ℓ = j
```

5. Algorithm for finding an actual 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 last pile, which is the last element of a longest increasing subsequence. Thus we can repeat the modified algorithm n times (ignoring the elements larger than or equal to the one output last) and find an actual longest increasing subsequence.[*2] The running time will be $O(\frac{1}{s} n^3 \log n)$.

As we claimed before we can do much better. We need only a $\log n$ multiplicative factor instead of n.

Theorem 5.1. *Let s be an integer satisfying $\sqrt{n} \le s \le n$. Given a permutation π of $\{1, \ldots, n\}$, a longest increasing subsequence of π can be found in $O(\frac{1}{s} n^2 \log^2 n)$ time using $O(s \log n)$ bits of additional space.*

Here we only give the high level idea. In the algorithm, we first find the $\lfloor \mathsf{lis}(\pi)/2 \rfloor$th element of a longest increasing subsequence. This can be done by simultaneously running the algorithm in the previous section normally and reversely (i.e., finding a longest decreasing subsequence from right to left). We then divide the permutation into two parts at the element and recurse. We can show that the depth of recursion is $O(\log n)$ and at each depth the total running time is $O(\frac{1}{s} n^2 \log n)$. To remember the current recursion, we need some additional space, but it is bounded by $O(\log^2 n)$ bits.

6. Lower bound for multi-pass algorithms

An algorithm is a *multi-pass* algorithm if it can access elements in the input array only by scanning the array from the beginning. For example, let i be the position of the element we want to read and j be the current position of the pointer on the input array. If $i \le j$, we need $j - i$ steps to move the pointer forward to read the element i. If $i > j$, we need to scan the input array from the beginning and need i steps to read the element.

It is easy to see that our algorithm for computing $\mathsf{lis}(\pi)$ is a multi-pass algorithm. Note that our algorithm for finding an actual longest increasing subsequence is not a multi-pass algorithm.

The following theorem can be shown by modifying a lower bound [5] for streaming algorithms computing $\mathsf{lis}(\pi)$.

Theorem 6.1. *Given a permutation π of $\{1, \ldots, n\}$, any multi-pass algorithm computing $\mathsf{lis}(\pi)$ using b bits takes $\Omega(n^2/b)$ time.*

Thus our multi-pass algorithm has optimal running time up to a polylog factor of $\log^2 n$.

References

[1] D. Aldous and P. Diaconis. Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem. *Bulletin of the American Mathematical Society*, 36(4):413–432, 1999.
[2] S. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76(1-2):7–11, 2000.
[3] A. Burstein and I. Lankham. Combinatorics of patience sorting piles. *Séminaire Lotharingien de Combinatoire*, 54A:B54Ab, 2006.
[4] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975.
[5] P. Gopalan, T. Jayram, R. Krauthgamer, and R. Kumar. Estimating the sortedness of a data stream. In *SODA*, pages 318–327, 2007.
[6] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.
[7] P. 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.
[8] M. Saks and C. Seshadhri. Space efficient streaming algorithms for the distance to monotonicity and asymmetric edit distance. In *SODA*, pages 1698–1709, 2013.
[9] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of computer and system sciences*, 4(2):177–192, 1970.
[10] C. Schensted. Longest increasing and decreasing subsequences. *Canad. J. Math*, 13(2):179–191, 1961.

[*2] This algorithm outputs a longest increasing subsequence in the reversed order. One can virtually reverse the input and find a longest decreasing subsequence to avoid this inconvenience.