Simpler Optimal Sorting from a Directed Acyclic Graph

Ivor van der Hoog* Eva Rotenberg* Daniel Rutschmann*

Abstract

Fredman proposed in 1976 the following algorithmic problem: Given are a ground set X, some partial order P over X, and some comparison oracle O_L that specifies a linear order L over X that extends P. A *query* to O_L has as input distinct $x, x' \in X$ and outputs whether $x <_L x'$ or vice versa. If we denote by $e(P)$ the number of linear orders that extend P, then it follows from basic information theory that $\log e(P)$ is a worst-case lower bound on the number of queries needed to output the sorted order of X.

Fredman did not specify in what form the partial order is given. Haeupler, Hladík, Iacono, Rozhon, Tarjan, and Tětek ('24) propose to assume as input a directed acyclic graph, G, with m edges and $n = |X|$ vertices. Denote by P_G the partial order induced by G. Their algorithmic performance is measured in running time and the number of queries used, where they use $\Theta(m + n + \log e(P_G))$ time and $\Theta(\log e(P_G))$ queries to output X in its sorted order. Their algorithm is worst-case optimal, both in terms of running time and queries. Their analysis relies upon sophisticated counting arguments using entropy, recursively defined sets defined over the run of their algorithm, and vertices in the graph that they identify as bottlenecks for sorting.

We do away with sophistication. We show that when the input is a directed acyclic graph then the problem admits a simple solution using $\Theta(m + n + \log e(P_G))$ time and $\Theta(\log e(P_G))$ queries. Especially our proofs are much simpler as we avoid the usage of advanced charging arguments, and instead rely upon two observations.

Funding. Ivor van der Hoog has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 899987. Eva Rotenberg and Daniel Rutschmann are supported by Eva Rotenberg's Carlsberg Foundation Young Researcher Fellowship CF21-0302 – "Graph Algorithms with Geometric Applications".

1 Introduction

Sorting is a fundamental problem in computer science. In 1976, Fredman introduced a natural generalisation of sorting, namely sorting under partial information [4]: Given a ground set X of size n, and some partial order P on X, and an oracle O_L with access to a ground-truth linear order L of X, the task is to minimise the number of oracle queries to recover the linear order of X. A query takes two input elements from X, and outputs their relation in the linear order. Algorithmic efficiency is measured in time, and the number of queries used. Let $e(P)$ denote the number of linear orders that extend the partial order P. Then, for any binary (e.g. yes/no) query, $\lceil \log_2(e(P)) \rceil$ is a worst-case lower bound for the number of queries needed, simply because the sequence of answers should be able to lead to any possible extension of the order.

Previous work. Fredman [4, TCS'76] shows an exponential-time algorithm for sorting under partial information that uses $\log e(P) + O(n)$ queries. This fails to match the $\Omega(\log e(P))$ lower bound when $\log e(P)$ is sublinear. Kahn and Saks [9, Order'84] prove that there always exists a query which reduces the number of remaining linear extensions by a constant fraction; showing that an $O(\log e(P))$-query algorithm exists. Kahn and Kim [8, STOC'92] are the first to also consider algorithmic running time. They note that Kahn and Saks [9] can preprocess a partial order using exponential time and space, so that given O_L it can output the sorted order of X in linear time plus $O(\log e(P))$ queries. Kahn and Kim [8, STOC'92] propose the first polynomial-time algorithm that performs $O(\log e(P))$ queries. They do not separate preprocessing the partial order and oracle queries, and use an unspecified polynomial time using the ellipsoid method. Cardinal, Fiorini, Joret, Jungers and Munro [3, STOC'10] (in their full version in Combinatorica) preprocess P in $O(n^{2.5})$ time, to output the sorted X in $O(\log e(P) + n)$ time using $O(\log e(P))$ queries. Their runtime poses the question whether query-optimal (sub)quadratic-time algorithms are possible.

*Department of Applied Mathematics and Computer Science, Technical University of Denmark, Denmark

For subquadratic algorithms, it becomes relevant how one obtains the partial order. Van der Hoog, Kostityna, Löffler and Speckmann [11, SOCG'19] study the problem in a restricted setting where P is induced by a set of intervals. Here, distinct $x_i, x_j \in X$ are incomparable whenever their intervals $([a_i, b_i], [a_j, b_j])$ intersect (otherwise, $x_i \prec_P x_j$ whenever $b_i < a_j$). They preprocess the intervals in $O(n \log n)$ time such that, given O_L, they can produce a pointer to a linked list storing X in its sorted order using $O(\log e(P))$ queries and time. For their queries, they use *finger search* [2] (i.e., *exponential search* from a pointer). Finger search has as input a value x_i, a sorted list π, and a finger $p \in \pi$ with $p < x_i$. It finds the farthest q along π for which $q < x_i$ in $O(1 + \log d_i)$ time and comparisons. Here, d_i denotes the length of the sublist from p to q [2].

Van der Hoog and Rutschmann [7, FOCS'24] assume that P is given as another oracle O_P where queries receive distinct $x, x' \in X$ and output whether $x \prec_P x'$. For fixed $c \geq 1$, they preprocess O_P using $O(n^{1+\frac{1}{c}})$ time and queries to O_P. Given O_L, they produce a pointer to a linked list storing X in its sorted order using $O(c \cdot \log e(P))$ queries and time. Their query algorithm also makes use of finger search. They show matching lower bounds in their setting.

Haeupler, Hladík, Iacono, Rozhon, Tarjan, and Tětek [5, '24] assume that the partial order P_G is induced by a graph G with m edges. Their algorithm can be seen as a combination of topological sort and heapsort. Their algorithm first isolates a collection $B \subset X$ which they call *bottlenecks*. They then iteratively consider all sources S in $G - B$, remove the source $s \in S$ that is *minimum* in the linear order L, and append s to the output π. Before appending s to π, they find the maximum prefix $B_s = \{b \in B \mid b <_L s\}$ using finger search where the finger is the head of B. They remove B_s from B, append B_s to π, and then append s. To obtain the minimum $s \in S$, they store all current sources of the graph in a *heap* where comparisons in the heap use the linear oracle O_L. Their algorithm does not separate preprocessing the graph G, and oracle queries. It uses $\Theta(n + m + \log e(P_G))$ overall time and $\Theta(\log e(P))$ linear oracle queries. Since reading the input takes at least $\Omega(n + m)$ time, their overall algorithmic runtime is thereby worst-case optimal. The bulk of their analysis is dedicated to charging the algorithmic runtime and query time to the workings of their heap, and to handling the special bottleneck vertices. We note that for any input k, their solution can also report the first k elements of X in optimal time.

Contribution. We consider the setting from [5], where the input is a directed acyclic graph G over X with m edges and where, unlike [3, 7, 11], one does not separate the algorithmic performance between a pre-processing phase using G and a phase that uses queries to O_L. We show that this problem formulation allows for a surprisingly simple solution: G denotes the input and H_i the graph at iteration i. Remove a maximum-length directed path π from G to get H_1. Iteratively remove an *arbitrary* source x_i from H_i to get H_{i+1}. Insert x_i into π using *finger search* where the finger p_i is the farthest in-neighbour of x_i in G along π (we find p_i by simply iterating over all in-neighbours of x_i). We use $\Theta(n + m + \log e(P_G))$ time and $\Theta(\log e(P_G))$ queries. We consider this algorithm to be simpler. Our proof of correctness is considerably simpler, as it relies upon only one counting argument and one geometric observation.

2 Algorithm

The input is a directed acyclic graph G with vertex set $X = (x_1, \ldots, x_n)$. This graph induces a partial order P_G where $x_i \prec x_j$ if and only if there exists a directed path from x_i to x_j in G. The input also contains an oracle O_L that specifies a linear order L over X. For any distinct x_i, x_j, an (oracle) query answers whether $x_i <_L x_j$. The goal is to output X in its sorted order L.

We describe our algorithm. Our key observation is that the problem becomes significantly easier when we first extract a maximum-length directed path π from G. We iteratively remove an *arbitrary* source x_i from $H = G - \pi$ and insert it into π. Once H is empty, we return π. In our analysis, our logarithms are base 2 and $d^*(x_i)$ denotes the sum of the out-degree and the in-degree of a vertex x_i in G.

Data structures. We maintain a path π where for all vertices $p, q \in \pi$: $p <_L q$ if and only if p precedes q in π. We store the path π in a dynamic list order structure T_π which is a data structure that supports the following:

- FingerInsert(q_i, x_i). Given vertices $x_i \notin \pi$ and $q_i \in \pi$, insert x_i into π succeeding q_i.
- Search(x_i, p_i, O_L). Given a vertex $x_i \notin \pi$ and a vertex $p_i \in \pi$ with $p_i <_L x_i$. Return the farthest vertex q_i along π where $q_i <_L x_i$. Let d_i denote the number of vertices on the subpath from p_i to q_i along π. We want to use $O(1 + \log d_i)$ time and queries to O_L.
- Compare(p, q). Given $p, q \in \pi$, return q if it succeeds p in π in $O(1)$ time (return p otherwise).

To achieve this, we store π in leaf-linked a balanced finger search tree T_π. Many implementations exist [2]. We choose the level-linked (2-4)-tree by Huddleston and Mehlhorn [6], which supports FingerInsert in amortised $O(1)$-time, and Search in $O(1 + \log d_i)$ time. For Compare, we store the linked leaves in the simple dynamic list ordering structure by Bender et al. [1] whose simplest implementation supports Compare in $O(1)$ time, and FingerInsert in amortised $O(1)$ time.

Algorithm and runtime analysis. Our algorithm is straightforward. The pseudo code is given by Algorithm 1 where each algorithmic step also indicates the running time. We require linear space. We first extract a longest directed path π from G, which we store in our two data structures. This takes $O(n + m)$ total time. If π has $n - k$ vertices, this leaves us with a graph H with k vertices.

We then iteratively remove an arbitrary source x_i from H. Since x_i is a source in H, all vertices $p \in X$ that have a directed path to x_i in G must be present in π. We iterate over all in-neighbours of x_i in G, and use our Compare operation to find the in-neighbour p_i that is furthest along π. If x_i has no in-neighbours, p_i is a dummy vertex that precedes the head of π instead. Finally, we proceed in a way that is very similar to the algorithm for topological sorting by Knuth [10]: We remove x_i from H, iterate over all out-neighbours of x_i in H and decrement their in-degree, and update the linked list of sources to include the newly found ones. Since we inspect each in- and out-edge of x_i only once, this takes $O(m)$ total time.

We then want to find the farthest q_i along π that precedes x_i in L. In the special case where p_i is the dummy vertex preceding the head h of π, we use a query to O_L to check whether $x_i <_L h$. If so, we return $q_i = p_i$. Otherwise, we set $p_i = h$ and invoke Search. This returns q_i in $O(1 + \log d_i)$ time and queries, where d_i is the length of the subpath from p_i to q_i in π ($d_i = 1$ if $p_i = q_i$).

Finally, we insert x_i into T_π succeeding q_i in constant time. Updating the dynamic list order data structure and rebalancing T_π takes amortised $O(1)$ time. Thus, these operations take $O(n)$ total time over all insertions. Therefore, our algorithm uses $O\Big(n + m + \sum_{x_i \in H} (1 + \log d_i)\Big)$ time and $O\Big(\sum_{x_i \in H} (1 + \log d_i)\Big)$ queries.

Algorithm 1 Sort(directed acyclic graph G over a ground set X, Oracle O_L) — time

1: $\pi \leftarrow$ a longest directed path in G ▷ $O(n + m)$
2: $T_\pi \leftarrow$ a level-linked (2-4)-tree over π [6] ▷ $O(n)$
3: $H \leftarrow G - \pi$ ▷ $O(n + m)$
4: Compute for each vertex in H its in-degree in H ▷ $O(n + m)$
5: $S \leftarrow$ sources in H ▷ $O(n)$
6: **while** $S \neq \emptyset$ **do**
7: Remove an arbitrary vertex x_i from S ▷ $O(1)$
8: $p_i \leftarrow$ a dummy vertex, which is prepended before the head of π ▷ $O(1)$
9: **for all** in-neighbors u of x_i in G **do** ▷ $O(d^*(x_i))$
10: $p_i \leftarrow$ Compare(p_i, u) ▷ $O(1)$
11: Remove x_i from H and add any new sources in H to S ▷ $O(d^*(x_i))$
12: $q_i \leftarrow$ Search(x_i, p_i, O_L) ▷ $O(1 + \log d_i)$
13: FingerInsert(q_i, x_i) ▷ $O(1)$ amortised
14: **return** the leaves of T_π in order ▷ $O(n)$

2.1 Proof of optimality.

Recall that $e(P_G)$ denotes the number of linear extensions of P_G and let H start out with k vertices. For ease of analysis, we re-index the vertices X so that the path π are vertices $(x_{k+1}, \ldots, x_n)$, in order, and x_i for $i \in [k]$ is the i'th vertex our algorithm inserts into T_π. We prove that our algorithm is tight by showing that $\sum_{i=1}^{k} (1 + \log d_i) = k + \sum_{i=1}^{k} \log d_i \in O(\log e(P_G))$.

Lemma 2.1. *Let G be a directed acyclic graph, P_G be its induced partial order and π be a longest directed path in G. If π has $n - k$ vertices then* $\log e(P_G) \geq k$.

Proof. Denote $\ell(x)$ the length of the longest directed path in G from a vertex x. Denote for $i \in [n]$ by $L_i := |\{x \in X \mid \ell(x) = i\}|$. For each $i \in [n - k]$, there is one $u \in \pi$ with $\ell(u) = i$ so:

$$k = \sum_{i=1}^{n-k} (L_i - 1). \tag{2.1}$$

Consider a linear order L' that is obtained by first sorting all $x \in X$ by $\ell(x)$ from high to low, and ordering (u, v) with $\ell(u) = \ell(v)$ arbitrarily. The order L' must extend P_G, since any vertex w that has a directed path towards a vertex u must have that $\ell(w) > \ell(u)$. We count the number of distinct L' that we obtain in this way to lower bound $e(P_G)$:

$$e(P_G) \geq \prod_{i=1}^{n-k} L_i! \quad \Rightarrow \quad e(P_G) \geq \prod_{i=1}^{n-k} 2^{L_i - 1} = 2^{\left(\sum_{i=1}^{n-k} (L_i - 1)\right)} \quad \Rightarrow \quad e(P_G) \geq 2^k$$

Here, the first implication uses that $x! \geq 2^{x-1}$ and the second implication uses Equation 2.1. ∎

What remains is to show that $\sum_{i=1}^{k} \log d_i \in O(\log e(P_G))$. To this end, we create a set of intervals:

Definition 1. *Let π^* be the directed path that Algorithm 1 outputs. For any $i \in [n]$ denote by $\pi^*(x_i)$ the index of x_i in π^*. We create an embedding E of X by placing x_i at position $\pi^*(x_i)$.*

Recall that for $i \in [k]$, p_i is the finger from where Algorithm 1 invokes Search *with x_i as the argument. We create as set $\mathcal{R}$ of n open intervals $R_i = (a_i, b_i) \subseteq [0, n]$ as follows:*

- *If $i > k$ then $(a_i, b_i) := (\pi^*(x_i) - 1, \pi^*(x_i))$.*
- *Else, $(a_i, b_i) := (\pi^*(p_i), \pi^*(x_i))$.*

Lemma 2.2. *Given distinct $x_i, x_j \in X$. If there exists a directed path from x_i to x_j in G then the intervals $R_i = (a_i, b_i)$ and $R_j = (a_j, b_j)$ are disjoint with $b_i \leq a_j$.*

Proof. We consider three cases.

If $i > k$ and $j > k$ then x_i and x_j are part of the original longest directed path in G. If x_i has a directed path to x_j in G then x_i precedes x_j on the original longest directed path. Thus, $\pi^*(x_i)$ precedes $\pi^*(x_j)$ and R_i and R_j are distinct unit intervals where R_i precedes R_j.

If $i \leq k$ and $j > k$ then x_i was inserted into the path π with x_j already in π. Since π^* is a linear extension of P_G, it follows that x_i was inserted preceding x_j in π. It follows that x_i precedes x_j in π^* and thus $R_i = (\pi^*(p_i), \pi^*(x_i))$ lies strictly before $R_j = (\pi^*(x_j) - 1, \pi^*(x_j))$.

If $j \leq k$ then the vertex p_j must equal or succeed x_i in the path π^*. It follows that $b_i = \pi^*(x_i) \leq a_i = \pi^*(p_j)$. The fact that these intervals are open then makes them disjoint. ∎

The set $\mathcal{R}$ induces an interval order $P_{\mathcal{R}}$ over X, which is the partial order $\prec_{\mathcal{R}}$ where x_i and x_j are incomparable whenever R_i and R_j intersect, and where otherwise $x_i \prec_{\mathcal{R}} x_j$ whenever $b_i \leq a_j$. By Lemma 2.2, any linear order L that extends $P_{\mathcal{R}}$ must also extend P_G and so $e(P_{\mathcal{R}}) \leq e(P_G)$. The number of linear extensions of an interval order is much easier to count. In fact, Cardinal, Fiorini, Joret, Jungers and Munro [3] and Van der Hoog, Kostityna, Löffler and Speckmann [11] already upper bound the number of linear extensions of an interval order. We paraphrase their upper bound, to give a weaker version applicable to this paper. For a proof from first principles, see Section 3.

Lemma 2.3. (Lemma 1 in [11] and Lemma 3.2 in [3]) *Let $\mathcal{R} = (R_1, \ldots R_n)$ be a set of n open intervals in $[0, n]$ and let each interval have at least unit size. Let $P_{\mathcal{R}}$ be its induced partial order. Then:*

$$\sum_{i=1}^{n} \log(|R_i|) \in O(\log e(P_{\mathcal{R}})).$$

We are now ready to prove our main theorem:

THEOREM 2.1. *Given a directed acyclic graph G over X, inducing a partial order P_G, and an oracle O_L whose queries specify a linear order L that extends P_G, there exists an algorithm that uses linear space, $O(n+m+\log e(P_G))$ time and $O(\log e(P_G))$ oracle queries to output the sorted order of X.*

Proof. Our algorithm runs in $O(n+m+k+\sum_{i=1}^{k}\log d_i)$ time and uses $O(k+\sum_{i=1}^{k}\log d_i)$ queries. By Lemma 2.1, $k \in O(\log e(P_G))$. The set $\mathcal{R}$ of Definition 1 is a set where each interval R_i has at least unit size. By Lemma 2.2, $e(P_{\mathcal{R}}) \leq e(P_G)$. For $i \leq k$, the size $|R_i|$ must be at least d_i since there are per construction at least d_i vertices on the subpath from $\pi^*(p_i)$ to $\pi^*(q_j)$ in the embedding E. It follows by Lemma 2.3 that $\sum_{i=1}^{k}\log d_i \in O(\log e(P_{\mathcal{R}})) \subseteq O(\log e(P_G))$. ∎

3 Deriving Lemma 2.3 from first principles

We consider the following statement:

LEMMA 2.3. (LEMMA 1 IN [11] AND LEMMA 3.2 IN [3]) *Let $\mathcal{R} = (R_1, \dots R_n)$ be a set of n open intervals in $[0,n]$ and let each interval have at least unit size. Let $P_{\mathcal{R}}$ be its induced partial order. Then:*

$$\sum_{i=1}^{n}\log(|R_i|) \in O(\log e(P_{\mathcal{R}})).$$

While this lemma follows verbatim from Lemma 1 in [11] (which itself is weaker version of Lemma 3.2 in in [3]), we also give a proof from first principles. We reindex $\mathcal{R}$ so that $(R_1, R_2, \dots, R_m)$ is a maximum cardinality set of pairwise-disjoint intervals in R, sorted from left to right. For any $R_i \in \mathcal{R}$ we denote by $\text{mid}(R_i)$ its centre. We define an $(n-m)$-dimensional polytope associated with $\mathcal{R}$.

$$A = \left\{ x \in \mathbb{R}^n \middle| x_i = \text{mid}(R_i) \text{ for } i \leq m \text{ and } x_i \in R_i \text{ for } i > m \right\}$$

The volume of this polytope is $\text{Vol}(A) = \prod_{i=m+1}^{n}|R_i|$. Any $x = (x_1, \dots, x_n) \in A$ with distinct coordinates defines a linear extension L of $P_{\mathcal{R}}$ via the rule "$i \prec j$ if and only if $x_i < x_j$". We call the point x a *realisation of* L. Observe that not all linear extensions of $P_{\mathcal{R}}$ have a realisation $x \in A$.

CLAIM 1. *Let L be a linear extension of P_R. Let $A_L = \{x \in A | x \text{ realises } L\}$, then*

$$\text{Vol}(A_L) \leq (2e)^{n-m}.$$

Proof. Consider the $m+1$ open intervals:

$$I := (-0.5, \text{mid}(R_1)), \dots, (\text{mid}(R_i), \text{mid}(R_{i+1})), \dots, (\text{mid}(R_{m-1}), n+0.5).$$

For all $(i,j) \in [m] \times [m]$, the intervals (R_i, R_j) are disjoint and have length at least 1. It follows that each interval in I has length at least 1. Suppose that L cannot be realised, then $\text{Vol}(A_L) = 0$. Otherwise, we observe that for any x, x' realising L and any $i \in \{m+1, ..., n\}$, an interval in I contains x_i if and only if it contains x'_i. This allows us to say that an interval in I is *occupied under* L if it contains x_i for any realisation x of L and any $i > m$. There are at most $n-m$ occupied intervals, and therefore at least $m+1-(n-m) = (2m+1-n)$ non-occupied intervals in I. Let G_L be the union of the occupied intervals. Then $|G_L| \leq |(-0.5, n+0.5)| - (2m+1-n) = 2(n-m)$, as each non-occupied interval has length at least 1. Let:

$$B_L = \left\{ y \in \mathbb{R}^m \times G_L^{n-m} \middle| y_i = \text{mid}(R_i) \text{ for } i \leq m \text{ and } (x_j < x_k \Leftrightarrow y_j < y_k) \text{ for } j,k > m \right\}.$$

Then $A_L \subseteq B_L$, hence

$$\text{Vol}(A_L) \leq \text{Vol}(B_L) = \frac{|G_L|^{n-m}}{(n-m)!} = \frac{(2(n-m))^{n-m}}{(n-m)!} \leq (2e)^{n-m}.$$

Which proves the claim. ∎

Claim 2. $\sum_{i=m+1}^{n} \ln |R_i| \leq \ln e(P_R) + (1 + \ln 2)(n - m)$

Proof. We note that:

$$\sum_{i=m+1}^{n} \ln |R_i| = \ln \mathrm{Vol}(A) \leq \ln \sum_{L \supseteq P_R} \mathrm{Vol}(A_L).$$

We now use Claim 1 to upper bond $\mathrm{Vol}(A_L)$ for all L and the claim follows. ∎

Claim 3.

$$\sum_{i=1}^{m} \ln |R_i| \leq (n - m)$$

Proof. As $R_1, \ldots, R_m$ are pairwise disjoint intervals in $[0, n]$, $\sum_{i=1}^{m} |R_i| \leq n$.
By analysis, $\ln x \leq x - 1$, hence

$$\sum_{i=1}^{m} \ln |R_i| \leq \sum_{i=1}^{m} (|R_i| - 1) = (n - m).$$

Which proves the claim. ∎

Combining Claim 2 and Claim 3 yields

$$\sum_{i=1}^{n} \log |R_i| = \frac{1}{\ln(2)} \cdot \sum_{i=1}^{n} \ln |R_i| \leq \log e(P_R) + \left(1 + \frac{2}{\ln(2)}\right) \cdot (n - m).$$

We now apply Lemma 2.1 of our paper to note that $n - m \leq \log e(P_R)$ and Lemma 3 follows.

References

[1] Michael A Bender, Richard Cole, Erik D Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *European symposium on algorithms*, pages 152–164. Springer, 2002.

[2] Gerth Stølting Brodal. Finger search trees. In *Handbook of Data Structures and Applications*, pages 171–178. Chapman and Hall/CRC, 2018.

[3] Jean Cardinal, Samuel Fiorini, Gwenaël Joret, Raphaël M. Jungers, and J. Ian Munro. Sorting under partial information (without the ellipsoid algorithm). *Combinatorica*, 33(6):655–697, Dec 2013.

[4] Michael L. Fredman. How good is the information theory bound in sorting? *Theoretical Computer Science*, 1(4):355–361, April 1976.

[5] Bernhard Haeupler, Richard Hladík, John Iacono, Vaclav Rozhon, Robert Tarjan, and Jakub Tětek. Fast and simple sorting using partial information. *arXiv preprint arXiv:2404.04552*, 2024.

[6] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta informatica*, 17:157–184, 1982.

[7] Daniel Rutschmann Ivor van der Hoog. Tight bounds for sorting under partial information. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS*. IEEE, 2024.

[8] Jeff Kahn and Jeong Han Kim. Entropy and sorting. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of Computing*, STOC '92, pages 178–187, New York, NY, USA, July 1992. Association for Computing Machinery.

[9] Jeff Kahn and Michael Saks. Balancing poset extensions. *Order*, 1(2):113–126, June 1984.

[10] Donald E Knuth. *The Art of Computer Programming: Fundamental Algorithms, Volume 1*. Addison-Wesley Professional, 1997.

[11] Ivor van der Hoog, Irina Kostitsyna, Maarten Löffler, and Bettina Speckmann. Preprocessing Ambiguous Imprecise Points. In *35th International Symposium on Computational Geometry (SoCG 2019)*, volume 129, pages 42:1–42:16, 2019.