Near-Optimal and Efficient Encoding for Two-Dimensional Range Minimum Queries

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland

Adam Górkiewicz
Institute of Computer Science, University of Wrocław, Poland

Srinivasa Rao Satti
Department of Computer Science, Norwegian University of Science and Technology, Trondheim, Norway

Abstract

We consider the 2D RMQ encoding problem: given an $m \times n$ array of mn elements over a total order, encode it such that, for any query rectangle, the position of its maximum element can be reported without accessing the original array. For $m \leq n$, it is known how to encode the array in $\mathcal{O}(mn \min\{m, \log n\})$ bits with $\mathcal{O}(1)$-time queries [Brodal et al., Algorithmica 2012], and also how to obtain an asymptotically optimal encoding consisting of $\mathcal{O}(mn \log m)$ bits [Brodal et al., ESA 2013]. However, the latter approach does not prove any guarantee on the query time, and it appears to be inherently sequential: it requires scanning the whole encoding to answer a query. We design a different encoding that uses near-optimal space while allowing for efficient queries. More concretely, for every parameter $\kappa \in [1, \log \log n]$, our encoding uses $\mathcal{O}(\kappa mn(\log m + \log \log n))$ bits and answers 2D RMQ queries in $\mathcal{O}(\log^{1/\kappa} n)$ time.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Encoding, Range Minimum Queries, Compact structure

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.51

Related Version *Full Version*: https://arxiv.org/abs/2607.04509

Funding *Paweł Gawrychowski and Adam Górkiewicz*: Partially supported by the Polish National Science Centre grant number 2023/51/B/ST6/01505.

1 Introduction

Range minimum/maximum data structures are important substructures in the design of succinct and compressed data structures. The *range maximum query* (RMQ) problem[1] asks us to preprocess an array so that, given a query range, we can return the position of a maximum element in that range. In the one-dimensional setting, the input is an array of length n and a query is an interval $[i \,..\, j]$. In two dimensions, the input is an $m \times n$ array and a query is an axis-parallel rectangle $[i_1 \,..\, i_2] \times [j_1 \,..\, j_2]$.

The RMQ problem has been studied in the literature in two distinct models: *indexing* and *encoding* models [46]. In the indexing model, the input array remains available at query time, and the data structure stores only auxiliary information, referred to as the *index*, to support the queries efficiently. In the encoding model, which is more relevant to this paper, queries are answered without access to the original input, and hence only the information relevant to answer queries is encoded within the data structure. The motivation behind

[1] RMQ traditionally refers to range *minimum* queries; we phrase the problem in terms of maxima, which is more convenient for our presentation. The two variants are equivalent by negating all elements.

the encoding model is twofold. First, it allows us to better understand the mathematical properties of the particular queries, and bound their effective entropy. Second, in some applications we need to support queries on an implicitly defined input that is well-defined but not stored at all (and not easy to compute in the query time). There has been a significant amount of research in the recent past on designing indexes and encodings for various range queries on arrays, such as 1D RMQ [17, 18, 8, 50, 12, 19, 2, 23, 41, 40, 39, 35], 2D RMQ [1, 13, 52, 8, 7, 6, 27, 34, 36, 24], range min&max [25, 33, 51, 31], range median and range mode [38, 43, 44, 5, 15, 28, 9], range selection and range top-k [11, 29, 25, 32, 15], and range majority and range minority [37, 14, 10, 21, 3, 42, 26, 22]. See [34] for a survey on encodings for range queries on arrays.

RMQ is a basic primitive in data structures and algorithms. In one dimension, it is tightly connected to the lowest common ancestor (LCA) problem on trees, and has applications in suffix-array based text indexing, compressed suffix trees, document retrieval, geometric indexing and many others. In two dimensions, it is a special case of the orthogonal range-searching problem and is relevant as a subroutine in geometric indexing applications. The 1D problem is by now very well understood, with almost tight bounds known both in the indexing and the encoding models. On the other hand, the 2D problem is qualitatively more subtle: while almost optimal space indexes supporting constant-time queries are known, the encoding complexity is not well-settled. The gap between optimal encoding space and efficient query support for the 2D RMQ encoding problem remains one of the main structural differences between 1D and 2D RMQ, which is the problem we address in this paper.

1.1 Previous results

1D case. The 1D RMQ problem has been widely studied, and many linear-space (i.e., $\mathcal{O}(n \log n)$ bits) data structures with constant query-time have been proposed in the literature [4, 30, 49], most of them using the fact that RMQ is reducible to the LCA on the *Cartesian tree* of the input array. The index space is further reduced to $\mathcal{O}(n/c)$ bits while supporting queries in $\mathcal{O}(c)$ time [18], which is shown to be asymptotically optimal [8]. For the encoding model, the first breakthrough was due to Sadakane [48] who gave an encoding of size $4n + o(n)$ bits that supports queries in constant time. The space usage is reduced to the optimal $2n + o(n)$ bits by Fischer and Heun [18]. More recent work on 1D RMQ focused on the tradeoff between the query time and the lower order terms in the space usage: Liu [39] showed that any data structure supporting RMQ queries using $2n - 1.5 \log n + r$ bits and query time t, must satisfy $r = n/(\log n)^{\mathcal{O}(t \log^2 t)}$, which closely matches the encoding of Liu and Yu [40] that uses $2n - 1.5 \log n + n/(\frac{\log n}{t})^t$ bits to support queries in $\mathcal{O}(t)$ time, for any parameter $t > 1$. Thus the encoding complexity of 1D RMQ has been well-settled.

2D case. For an $m \times n$ input array, the multidimensional solution of Gabow et al. [20] gives a data structure that takes $\mathcal{O}(N \log^2 N)$ bits and supports queries in $\mathcal{O}(\log N)$ time, where $N = mn$. The subsequent results [45, 1, 52] improved these bounds to $\mathcal{O}(N \log N)$ bits and $\mathcal{O}(1)$ query time. Brodal et al. [8] designed the first linear-bit (i.e., $\mathcal{O}(N)$ bits) index that supports queries in constant time. They also proposed indexes that tradeoff query time for index size, which was further improved by [7].

For the encoding model, Demaine et al. [13] proved that there is no natural Cartesian-tree analogue for the 2D RMQ problem, and in particular showed a counting lower bound of $\Omega(n^2 \log n)$ bits for $n \times n$ arrays. Brodal et al. [8] extended the lower bound to $\Omega(mn \log m)$ bits for $m \times n$ arrays with $m \leq n$. They also gave an $\mathcal{O}(1)$-query-time encoding using $\mathcal{O}(mn \cdot \min\{m, \log n\})$ bits, which is optimal when $\log n = \Theta(\log m)$ but not in general.

The general encoding complexity was settled shortly afterwards by Brodal et al. [6], who obtained an encoding that takes $\Theta(mn \log m)$ bits, matching the lower bound. However, their encoding does not support queries efficiently, and answering one may require decoding essentially the whole representation. Thus, prior to our work, there is a large gap between two extremes for the 2D RMQ encodings:

- an encoding with $\mathcal{O}(1)$ query time using $\mathcal{O}(mn \min\{m, \log n\})$ bits, and
- an asymptotically optimal $\mathcal{O}(mn \log m)$-bit encoding without efficient query support.

1.2 Our results

Bridging this gap is the main motivation for our work. The natural question is whether one can approach the optimal $\Theta(mn \log m)$ space bound while still supporting fast queries. Our main result answers this positively by showing a trade-off between space and query time.

▶ **Theorem 1.** *For any integer $\kappa \in [1, \log \log n]$, there is a 2D-RMQ encoding for an $m \times n$ array that uses $\mathcal{O}(\kappa mn(\log m + \log \log n))$ bits and answers queries in $\mathcal{O}(\log^{1/\kappa} n)$ time.*

Two regimes of Theorem 1 are worth singling out. For every constant $\epsilon > 0$, taking κ to be a sufficiently large constant yields an encoding of $\mathcal{O}(mn(\log m + \log \log n))$ bits with queries in $\mathcal{O}(\log^{\epsilon} n)$ time; the space is asymptotically optimal whenever n is at most exponential in m. At the other extreme, taking $\kappa = \log \log n$ drives the query time down to $\mathcal{O}(1)$ at the cost of a multiplicative $\log \log n$ blowup in space.

Overview. Section 3 reduces every query to a comparison between two distinguished points of the query rectangle, and Section 4 formalises the structure behind this comparison: every point is assigned an *origin* in an auxiliary tree built over the columns, and the two points produced by the reduction always share enough structure in this tree for the comparison to be resolved efficiently. Section 5 carries out the encoding itself in two passes. The baseline version (Section 5.2) stores a small comparison structure at each node of the column tree and answers a query by repeatedly replacing one of the two current points with a substitute whose origin lies closer to the root; this matches Theorem 1 at $\kappa = 1$. The full version (Section 5.3) introduces a second tree, built over the depths of the column tree, that guarantees enough progress in each replacement to bound their number by its arity; tuning the arity yields the entire trade-off of Theorem 1. Finally, Section 6 constructs the two local primitives used by both encodings, by decomposing the active points of each node into $\mathcal{O}(m)$ *quarter-rows*, inside which the weights are monotone with respect to the column.

2 Preliminaries

For integers i, j we write $[i \,..\, j]$ to denote $\{i, i+1, \ldots, j\}$. Throughout the paper, the input is an $m \times n$ array with rows indexed from 1 to m, and columns indexed from 1 to n. A *point* is a pair $(i, j) \in [1 \,..\, m] \times [1 \,..\, n]$. We treat the input as a function w mapping each point to a distinct weight; for a point $p = (i, j)$, we write $w(p)$ for its weight, $\mathrm{row}(p) := i$, and $\mathrm{col}(p) := j$. For a set of points S, we write $\max_w(S)$ for the unique point $p \in S$ maximising $w(p)$. A *query rectangle* is a set of the form $R = [i_1 \,..\, i_2] \times [j_1 \,..\, j_2] \subseteq [1 \,..\, m] \times [1 \,..\, n]$, and the 2D RMQ problem asks, given such a rectangle R, to return the heaviest point in it, namely $\max_w(R)$.

We measure the space in bits and assume the standard word RAM model of computation, with word size $\Theta(\log n)$, when describing the query algorithms.

We make use of the following 1D RMQ encoding.

▶ **Lemma 2** ([18]). *For an array of length n with distinct values, there is an RMQ structure using $\mathcal{O}(n)$ bits that returns the position of the maximum in any interval in $\mathcal{O}(1)$ time.*

In our encodings, we use representations of sets supporting the following operations. Given a set $S \subseteq [1\,..\,U]$,

- $\mathrm{rank}_S(x)$ returns the number of elements of S that are not greater than x, and
- $\mathrm{select}_S(i)$ returns the i-th smallest element in S.

We will only query rank on elements $x \in S$ (i.e., only partial-rank support is required), which allows for an encoding with a smaller redundancy term compared to the general case.

▶ **Lemma 3** ([47]). *Let $S \subseteq [1\,..\,U]$ with $|S| = k$. There is an encoding using $\mathcal{O}(k+k\log(U/k))$ bits supporting rank (only for members of S), and select queries in $\mathcal{O}(1)$ time.*

We also use the Elias–Fano encoding for representing monotonic sequences:

▶ **Lemma 4** ([16]). *Let $S[1\,..\,k]$ be a non-decreasing sequence with values in $[1\,..\,U]$, with $k \le U$. There is an encoding using $\mathcal{O}(k + k\log(U/k))$ bits supporting access to any $S[t]$ in $\mathcal{O}(1)$ time. The non-increasing case is symmetric.*

Lemma 3 and Lemma 4 can be further refined to optimise the leading constants, but this introduces clutter and does not improve our final result.

Throughout the paper, the space of a structure is almost always obtained by summing entropy-like bounds of the form $k_i \log(U_i/k_i)$ coming from the two encodings above. To handle such sums uniformly, we will rely on the following inequality proven in Appendix A.

▶ **Lemma 5** (log-sum inequality). *For any positive numbers $k_1, \ldots, k_t$ and $U_1, \ldots, U_t$, with total sums $K \coloneqq \sum_{i=1}^{t} k_i$ and $U \coloneqq \sum_{i=1}^{t} U_i$,*

$$\sum_{i=1}^{t} k_i \log \frac{U_i}{k_i} \le K \log \frac{U}{K}.$$

3 Reducing a Query to Two Candidates

The first step of the construction reduces the query rectangle to comparing two points. Since storing enough information to compare arbitrary pairs is far too expensive, the reduction must produce points with some additional structure.

▶ **Definition 6** (visibility). *Let $p = (i, j)$ be a point and let $j' \in [1\,..\,n]$ be a column. We say that p is* visible from column j' *if $p = \max_w(\{i\} \times [\min\{j, j'\}\,..\,\max\{j, j'\}])$. Equivalently, along row i, the point p remains the heaviest when scanning from column j toward column j'.*

▶ **Lemma 7.** *There is a data structure using $\mathcal{O}(mn \log m)$ bits that, given a query rectangle R, returns in $\mathcal{O}(1)$ time two (not necessarily distinct) points $p, q \in R$ such that $\max_w(R) = \max_w(\{p, q\})$, p is visible from $\mathrm{col}(q)$, and q is visible from $\mathrm{col}(p)$.*

Proof. Build a 1D RMQ structure of Lemma 2 for every row and column. In addition, for every $k \ge 0$ and every starting row r, consider the row block $[r\,..\,r + 2^k - 1] \times [1\,..\,n]$. For each such block, define an auxiliary array of length n whose j-th entry corresponds to the heaviest point in column j restricted to the block, and store its 1D RMQ structure of Lemma 2. Over all blocks, the total space is $\mathcal{O}(mn \log m)$ bits.

Suppose $R = [i_1\,..\,i_2] \times [j_1\,..\,j_2]$ and let $h \coloneqq i_2 - i_1 + 1$. If h is a power of 2, we can answer the query in $\mathcal{O}(1)$ time using the 1D RMQ structure for the row block $[i_1\,..\,i_2] \times [1\,..\,n]$. Otherwise, let $R_{\mathrm{top}} \coloneqq [i_1\,..\,i_1 + 2^{\lfloor \log h \rfloor} - 1] \times [j_1\,..\,j_2]$ and $R_{\mathrm{bot}} \coloneqq [i_2 - 2^{\lfloor \log h \rfloor} + 1\,..\,i_2] \times [j_1\,..\,j_2]$ be two overlapping rectangles. Let $p \coloneqq \max_w(R_{\mathrm{top}})$ and $q \coloneqq \max_w(R_{\mathrm{bot}})$. Since the two blocks cover R, the maximum of R is the heavier of p and q.

To prove mutual visibility, consider $p = (i, j)$ and the column $j' = \mathrm{col}(q)$. If p was not visible from j', then some point (i, t) with t between j and j' would satisfy $w(i, t) > w(p)$. Since both j and j' lie in $[j_1 \,..\, j_2]$, the point (i, t) would belong to R_{top}, contradicting the maximality of p there. The proof for q is symmetric. ◀

By applying the above preprocessing, the RMQ problem is reduced to comparing the two mutually visible points p and q, called *candidates*, and returning the heavier one.

In principle, the ability to compare arbitrary pairs of points would reveal the total order of all mn entries and thus requires $\Omega(mn \log(mn))$ bits of space. The visibility condition provides the additional structure that makes the comparison problem compressible. The rest of the paper is therefore concerned only with comparing mutually visible pairs of points.

4 Active Points and Origins

We now introduce the main concepts of the data structure, and formulate exactly which pairs of points are going to be comparable by it. We then show that the pair of candidates produced by Lemma 7 satisfies the requirement.

Tree on columns. Let $\mathcal{T}$ be a complete binary tree over the columns $[1 \,..\, n]$. Each leaf of $\mathcal{T}$ corresponds to a column, with the leaves ordered from left to right. Every node $u \in \mathcal{T}$ corresponds to an interval, denoted as $\mathrm{span}(u) \subseteq [1 \,..\, n]$, consisting of the columns of the leaves in the subtree of u. In particular, the interval of the root is the whole $[1 \,..\, n]$, and the intervals of the two children of an internal node partition the interval of their parent into two parts of almost equal length. The depth of u in $\mathcal{T}$ is denoted as $\mathrm{depth}(u)$, where the root has depth 1.

▶ **Definition 8** (active points). *For an internal node $u \in \mathcal{T}$ with children u_1 and u_2, whose intervals are $\mathrm{span}(u_1) = [a_1 \,..\, b_1]$ and $\mathrm{span}(u_2) = [a_2 \,..\, b_2]$, we define $\partial u := \{a_1, b_1, a_2, b_2\}$. Note that some of these endpoints may coincide. For a leaf $u \in \mathcal{T}$, we define $\partial u := \{c\}$, where c is the unique column in $\mathrm{span}(u)$.*

We define the active set *of u, denoted A_u, as the set of all points in $[1 \,..\, m] \times \mathrm{span}(u)$ that are visible from at least one column in ∂u. A point p is said to be* active in u *if $p \in A_u$.*

The notion of active points is useful because "activeness" behaves monotonically down the tree: once a point is active in some node, it remains active on the path toward its column leaf.

▶ **Lemma 9.** *For every point p, the set of nodes u such that $p \in A_u$ forms a downward path in $\mathcal{T}$ ending at the leaf corresponding to column $\mathrm{col}(p)$.*

Proof. Suppose that $p = (i, j)$. If $p \in A_u$, then by definition $j \in \mathrm{span}(u)$. Hence every node u such that $p \in A_u$ lies on the root-to-leaf path ending at the leaf corresponding to column j.

It therefore suffices to show that this set of nodes is downward closed. Let u be an internal node such that $p \in A_u$, and let v be the child of u whose interval contains j. Choose a witness column $c \in \partial u$ such that p is visible from c.

If $c \in \partial v$, then the same witness shows that $p \in A_v$. Otherwise, let $c' \in \partial v$ be the endpoint of $\mathrm{span}(v)$ closest to c. Then the interval between j and c' is contained in the interval between j and c. Since p is visible from c, it is also visible from c', and hence $p \in A_v$.

It follows that the active set of p is a downward-closed subset of this root-to-leaf path. Since p is always active in the leaf for column j (it is trivially visible from its own column), the set is nonempty, and therefore forms a path ending at the leaf corresponding to column j. ◀

▶ **Definition 10** (origin). *For a point p, we define the* origin *of p, denoted* $\mathrm{orig}(p)$*, as the topmost node of the path given by Lemma 9.*

▶ **Definition 11** (co-active pair). *A pair of points p, q is* co-active *if there exists a node $u \in \mathcal{T}$ such that $p, q \in A_u$.*

The next lemma gives a more algorithmic characterization of co-activity. Although the definition only requires the existence of some witness node in which both points are active, one can always pick a canonical one: the deeper of the two origins.

▶ **Lemma 12.** *Let p and q be points such that* $\mathrm{depth}(\mathrm{orig}(p)) \leq \mathrm{depth}(\mathrm{orig}(q))$*. Then p and q are co-active if and only if p is active in* $\mathrm{orig}(q)$*.*

Proof. If p is active in $\mathrm{orig}(q)$, then both p and q belong to $A_{\mathrm{orig}(q)}$, so the pair is co-active. Conversely, suppose that p and q are co-active. Then there exists a node $u \in \mathcal{T}$ such that $p, q \in A_u$. By Lemma 9, both $\mathrm{orig}(p)$ and $\mathrm{orig}(q)$ are ancestors of u. Since $\mathrm{depth}(\mathrm{orig}(p)) \leq \mathrm{depth}(\mathrm{orig}(q))$, it follows that $\mathrm{orig}(q)$ is a descendant of $\mathrm{orig}(p)$. Hence $\mathrm{orig}(q)$ lies on the active path of p, and therefore p is active in $\mathrm{orig}(q)$. ◀

It can be directly verified that the pairs produced by the reduction of Lemma 7 are co-active.

▶ **Lemma 13.** *If p is visible from column* $\mathrm{col}(q)$ *and q is visible from column* $\mathrm{col}(p)$*, then p and q are co-active.*

Proof. If $\mathrm{col}(p) = \mathrm{col}(q)$, then both p and q are active in the leaf of $\mathcal{T}$ corresponding to this column, so the pair is co-active. Otherwise, let u be the lowest common ancestor in $\mathcal{T}$ of the leaves corresponding to columns $\mathrm{col}(p)$ and $\mathrm{col}(q)$. In that case u is an internal node, and one of the columns $c \in \partial u$ lies between $\mathrm{col}(p)$ and $\mathrm{col}(q)$. Since p is visible from $\mathrm{col}(q)$, it is also visible from every column between $\mathrm{col}(p)$ and $\mathrm{col}(q)$, and in particular from c. Hence $p \in A_u$. By the same argument, $q \in A_u$. Therefore the pair is co-active. ◀

In the remainder of the paper, we focus on encodings capable of efficiently comparing precisely the co-active pairs of points. Combined with the preprocessing of Lemma 7, which reduces every RMQ query to such a comparison by Lemma 13, this yields the main result.

5 Comparing Co-Active Pairs

From this point onward, we consider the following comparison problem: given two co-active points p_0 and q_0, determine which of them is heavier. We work with its decision version, fixing the hypothesis $\mathcal{H} : w(p_0) \leq w(q_0)$, so that the answer is q_0 if $\mathcal{H}$ holds, and p_0 otherwise.

At a high level, the query algorithm maintains a co-active pair of points, initially (p_0, q_0), along with the invariant that the hypothesis "the first point is no heavier than the second" is equivalent to $\mathcal{H}$. At each step, one of the two current points gets replaced by a substitute whose origin in $\mathcal{T}$ is strictly closer to the root, in such a way that the invariant is preserved. Note that this strategy does not preserve the identity of the heavier point of the pair. The process terminates once a single local comparison can resolve the query.

5.1 Local Primitives

We now state the two local primitives that support the process described above; their implementation is deferred to Section 6. The first primitive is a *ranking* structure: given a set of points, it answers rank queries among them, and is used to perform the single comparison

that resolves the query at the end of the process. The second primitive is a *lifting* structure: it takes a point and returns a suitable substitute whose origin is closer to the root of $\mathcal{T}$, and is used to carry out the intermediate replacement steps.

Ranks of points. For a set of points L and a point $p \in L$, we define the *rank* of p in L as the number of points in L not heavier than p. In particular, knowing the ranks of two points in L immediately tells us which of them is heavier.

▶ **Lemma 14.** *Let u be a node of $\mathcal{T}$ with $\lambda_u \coloneqq |\operatorname{span}(u)|$. For every nonempty set $L \subseteq A_u$, there exists an encoding using*

$$\mathcal{O}\left(|L| \log m + |L| \log \frac{m\lambda_u}{|L|}\right)$$

bits that, given a point $p \in L$, returns in $\mathcal{O}(1)$ time the rank of p in L.

The intended use of Lemma 14 is simple: if we have built the encoding for some subset $L \subseteq A_u$ and both points of the current pair happen to lie in L, the decision problem can be resolved immediately by comparing their ranks in L.

Predecessors and successors. For a set of points V and a point p, the *predecessor* of p in V is the heaviest point in $\{q \in V : w(q) \le w(p)\}$, and the *successor* of p in V is the lightest point in $\{q \in V : w(p) \le w(q)\}$. If the first set is empty, the predecessor is defined to be $\bot$, and if the second set is empty, the successor is defined to be $\top$, whose weights are understood to be $-\infty$ and $+\infty$, respectively.

▶ **Observation 15.** *Let p, q be points, and let P, Q be such that $p \in P$ and $q \in Q$. Let p' be the successor of p in Q, and let q' be the predecessor of q in P. Then*

$$w(p) \le w(q) \iff w(p') \le w(q) \iff w(p) \le w(q').$$

Thus, provided that the other point belongs to the target set, one may replace one of the current points with an appropriate predecessor or successor, while preserving the invariant.

▶ **Lemma 16.** *Let u be a node of $\mathcal{T}$ with $\lambda_u \coloneqq |\operatorname{span}(u)|$. For every nonempty source set $S \subseteq A_u$ and target set $T \subseteq A_u$, there exists an encoding using*

$$\mathcal{O}\left(|S| \log m + |S| \log \frac{m\lambda_u}{|S|}\right)$$

bits that, given a point $p \in S$, returns in $\mathcal{O}(1)$ time the predecessor and successor of p in T.

The remainder of the paper is devoted to choosing suitable sets for Lemmas 14 and 16. The warm-up structure uses one lifting set per node of $\mathcal{T}$ and therefore raises the origin by one or more levels at a time. The faster structure stores several such sets, organised by a tree over the depth range of $\mathcal{T}$.

5.2 The Baseline Structure

We first describe a simpler $\mathcal{O}(mn(\log m + \log\log n))$-bit encoding with $\mathcal{O}(\log n)$ query time before we move to the full trade-off.

Stored structures. For every point p, we store $\mathrm{depth}(\mathrm{orig}(p))$. Note that $\mathrm{orig}(p)$ is uniquely determined by $\mathrm{col}(p)$ and $\mathrm{depth}(\mathrm{orig}(p))$, as it is the unique node of $\mathcal{T}$ at that depth whose interval contains $\mathrm{col}(p)$, by Lemma 9. We assume standard constant-time navigation in $\mathcal{T}$, which can be implemented with bitwise operations.

For every node $u \in \mathcal{T}$, we define S_u as the set of points in A_u whose origin is u itself, and T_u as those whose origin is a strict ancestor of u; by Lemma 9, these two sets partition A_u. For each such node u, we store the following two structures:

- by Lemma 14, a local ranking structure for the set S_u, and
- by Lemma 16, a lifting structure with source set S_u and target set T_u.

The idea is as follows. If the two current points have the same origin u, then both belong to S_u, and the ranking structure stored for u resolves the comparison. Otherwise, let u be the deeper of the two origins; then the corresponding point belongs to S_u, while the other point is also active in u with its origin strictly above, and hence belongs to T_u. In this case, the lifting structure stored for u is used. We now formalize this.

Query algorithm. Let R be the query rectangle. Using Lemma 7, we obtain two candidate points $p_0, q_0 \in R$ such that $\max_w(R)$ is the heavier of the two. By Lemma 13, the pair (p_0, q_0) is co-active. We fix the hypothesis $\mathcal{H} : w(p_0) \le w(q_0)$ and maintain an ordered pair (p, q), initially (p_0, q_0), with the invariant that the statement $w(p) \le w(q)$ is equivalent to $\mathcal{H}$.

If $\mathrm{orig}(p) = \mathrm{orig}(q) = u$, then $p, q \in S_u$, and the ranking structure stored for u resolves the comparison immediately. From now on assume that the two origins are different.

Suppose first that $\mathrm{depth}(\mathrm{orig}(q)) > \mathrm{depth}(\mathrm{orig}(p))$, and let $u \coloneqq \mathrm{orig}(q)$. By Lemma 12, we have $p \in A_u$, and hence $p \in T_u$. In this case we use the lifting structure stored for u to obtain q', the predecessor of q in T_u. By the invariant, the statement $w(p) \le w(q)$ is equivalent to $\mathcal{H}$, and since $p \in T_u$, Observation 15 tells us that the same holds for $w(p) \le w(q')$. At this point, either $q' = \bot$, in which case the hypothesis is false and the query is resolved immediately, or else $q' \in T_u$, and we replace q with q'. In the latter case, the origin of q' lies strictly above u, thus the origin of the deeper point has moved upward. Also, since both p and q' belong to A_u, the new pair (p, q') is again co-active.

The case $\mathrm{depth}(\mathrm{orig}(p)) > \mathrm{depth}(\mathrm{orig}(q))$ is symmetric: we now take $u \coloneqq \mathrm{orig}(p)$, and use the lifting structure stored for u to replace p with its *successor* p' in T_u, where $q \in T_u$ by Lemma 12. As before, Observation 15 preserves the invariant, and the two possible outcomes mirror those of the previous case: either $p' = \top$, which resolves the query, or we get a substitute point $p' \in T_u$, with origin strictly above u, and a new co-active pair (p', q).

The algorithm continues in this way until the two current points have the same origin, at which point the ranking structure finishes the comparison. Since the height of $\mathcal{T}$ is $\mathcal{O}(\log n)$, there can be at most $\mathcal{O}(\log n)$ such steps, and all work performed in each step is constant-time. Hence the total query time is $\mathcal{O}(\log n)$.

Space analysis. Most of the space analysis is taken care of by the following lemma, which we state as a self-contained bound. Its statement is tailored to match the per-node space bound of Lemmas 14 and 16, so that applying it reduces to verifying that the relevant families of sets are pairwise disjoint. Isolating the statement in this way also lets us reuse the same bound later in the space analysis of the full structure.

▶ **Lemma 17.** *Suppose that with every node $u \in \mathcal{T}$ we associate a set of points X_u, and that the family $\{X_u : u \in \mathcal{T}\}$ is pairwise disjoint. If for every u with $X_u \neq \emptyset$ we store a structure in $\mathcal{O}(|X_u| \log m + |X_u| \log(m\lambda_u/|X_u|))$ bits, where $\lambda_u \coloneqq |\operatorname{span}(u)|$, then the total space of all these structures is $\mathcal{O}(mn \log m + mn \log\log n)$ bits.*

Proof. We split the per-node bound into its two summands and handle them separately. For the first summand, since the sets X_u are pairwise disjoint subsets of $[1\,..\,m] \times [1\,..\,n]$, we have $\sum_u |X_u| \le mn$, and hence

$$\sum_u |X_u| \log m \le mn \log m.$$

It remains to bound $\sum_u |X_u| \log(m\lambda_u/|X_u|)$ by $\mathcal{O}(mn \log\log n)$.

Let H be the height of $\mathcal{T}$ (so $H = \mathcal{O}(\log n)$). For every depth $d \in [1\,..\,H]$, let $\mathcal{N}_d$ be the set of nodes u at depth d with $X_u \neq \emptyset$, and define

$$K_d := \sum_{u \in \mathcal{N}_d} |X_u|.$$

Since the sets X_u are pairwise disjoint, we have $\sum_{d=1}^{H} K_d \le mn$.

Fix a depth d with $K_d > 0$. The nodes of $\mathcal{T}$ at depth d have pairwise disjoint intervals, so $\sum_{u \in \mathcal{N}_d} m\lambda_u \le mn$. Applying Lemma 5 to the values $k_u := |X_u|$ and $U_u := m\lambda_u$ for $u \in \mathcal{N}_d$, we obtain

$$\sum_{u \in \mathcal{N}_d} |X_u| \log \frac{m\lambda_u}{|X_u|} \le K_d \log \frac{mn}{K_d}.$$

Now summing over all depths with $K_d > 0$ and applying Lemma 5 once again, this time to the values K_d with $U_d = mn$ for every such d, we get the total cost bound

$$\sum_{\substack{1 \le d \le H \\ K_d > 0}} K_d \log \frac{mn}{K_d} \le mn \log \frac{H \cdot mn}{mn} = mn \log H = \mathcal{O}(mn \log\log n). \qquad \blacktriangleleft$$

We now sum the space over all components. For every point p, the value $\mathrm{depth}(\mathrm{orig}(p))$ belongs to $[1\,..\,H]$, where $H = \mathcal{O}(\log n)$ is the height of $\mathcal{T}$, and thus can be stored in $\mathcal{O}(\log\log n)$ bits per point, for a total of $\mathcal{O}(mn \log\log n)$ bits.

Next, we bound the total space of the local ranking structures. For every node $u \in \mathcal{T}$ with $S_u \neq \emptyset$, we store one such structure on the set S_u, and its per-node cost, given by Lemma 14, coincides exactly with the one assumed in Lemma 17 (with $X_u = S_u$). Since every point has exactly one origin, the family $\{S_u : u \in \mathcal{T}\}$ partitions the set of all points, and in particular is pairwise disjoint. Hence Lemma 17 applies and bounds the total space of the ranking structures by $\mathcal{O}(mn \log m + mn \log\log n)$ bits.

The same argument applies to the lifting structures: each is built on S_u as its source set, and the per-node bound in Lemma 16 is identical to that of Lemma 14. Their total space is therefore also $\mathcal{O}(mn \log m + mn \log\log n)$ bits.

Finally, the reduction structure of Lemma 7 uses $\mathcal{O}(mn \log m)$ bits, which completes the proof, as all components fit within $\mathcal{O}(mn(\log m + \log\log n))$ bits of space.

5.3 The Full Structure

We now describe the general construction achieving the trade-off from Theorem 1. As in the baseline structure, the query algorithm fixes a comparison hypothesis and repeatedly replaces one of the two current points. The difference is that we no longer move one level of depth at a time in the tree $\mathcal{T}$. Instead, we group consecutive depths of $\mathcal{T}$ into larger blocks, and each lifting step moves the deeper point from its current block into an earlier one. The local ranking structures are also adapted: rather than comparing only points with exactly the same origin, they compare points whose origins lie in one prescribed depth block. As in the baseline structure, we store $\mathrm{depth}(\mathrm{orig}(p))$ for every point p.

Tree on depths. Fix a parameter $\kappa \in [1, \log\log n]$ and let $\tau \coloneqq \lceil \log^{1/\kappa} n \rceil$. Recall that the height of the complete binary tree $\mathcal{T}$ is $H = \mathcal{O}(\log n)$. We build a complete τ-ary tree $\mathcal{D}$ over the depths $[1\,..\,H]$ by recursively partitioning every interval into at most τ consecutive sub-intervals of almost equal lengths, with the children of every node ordered so that smaller depths lie to the left. For every node $x \in \mathcal{D}$, let $\mathrm{span}(x) \subseteq [1\,..\,H]$ denote the depth interval represented by x. The height of $\mathcal{D}$ is $\mathcal{O}(\kappa)$.

The tree $\mathcal{D}$ is an auxiliary structure used to organise the query process. We assume standard constant-time navigation in both $\mathcal{D}$ and $\mathcal{T}$, which can be implemented with bitwise operations (for $\mathcal{D}$ this requires rounding up τ to the nearest power of two).

Given two points, their origin depths determine two leaves of $\mathcal{D}$, and hence their lowest common ancestor node, whose children separate them. The algorithm will work relative to this one node of $\mathcal{D}$: it repeatedly moves the deeper origin across its children until both current origin depths lie in the interval of the same child. At that point a local ranking structure finishes the comparison.

Ranking structures. For every depth $d \in [1\,..\,H]$, let C_d denote the largest *canonical* interval of $\mathcal{D}$ of the form $[a\,..\,d]$, where an interval is canonical if it appears as $\mathrm{span}(x)$ for some node $x \in \mathcal{D}$. Now let $u \in \mathcal{T}$ be any node of depth d. We define

$$L_u \coloneqq \{p \in A_u : \mathrm{depth}(\mathrm{orig}(p)) \in C_d\}\,.$$

For every node $u \in \mathcal{T}$, we store one local ranking structure of Lemma 14 for the set L_u.

Lifting structures. We next describe the lifting structures. Let y be a non-leftmost child of a node $x \in \mathcal{D}$. If we denote $\mathrm{span}(x) = [a\,..\,b]$ and $\mathrm{span}(y) = [a'\,..\,b']$, then the depths lying in child intervals of x strictly to the left of y are precisely those in $[a\,..\,a'-1]$.

Now let $u \in \mathcal{T}$ be a node such that $\mathrm{depth}(u) \in \mathrm{span}(y)$. We define

$$T_{y,u} \coloneqq \{p \in A_u : \mathrm{depth}(\mathrm{orig}(p)) \in [a\,..\,a'-1]\}\,.$$

In words, $T_{y,u}$ consists of the points that are active in u, whose origin depths lie in one of the intervals of the left siblings of y. For every such pair (y, u), we store a lifting structure of Lemma 16 with source set S_u (the points with origin u) and target set $T_{y,u}$.

This is a natural generalisation of the baseline structure: instead of lifting a point with origin u from $\mathrm{depth}(u)$ to *any* smaller depth, we now lift it only into the union of the child intervals of x that lie to the left of the one currently containing $\mathrm{depth}(u)$. In particular, for a fixed node $u \in \mathcal{T}$, the sets $T_{y,u}$ partition the set T_u (points active in u whose origins lie strictly above u) according to the first branching node in $\mathcal{D}$ at which their origin depths diverge.

Query algorithm. Let R be the query rectangle. As before, Lemma 7 returns two candidate points $p_0, q_0 \in R$, and Lemma 13 guarantees that this pair is co-active. We fix the hypothesis $\mathcal{H} : w(p_0) \le w(q_0)$ and maintain an ordered pair (p, q), initially (p_0, q_0), with the same invariant as in the baseline structure.

If the two initial points have the same origin u, then both belong to L_u, since both are in A_u and $\mathrm{depth}(u) \in C_{\mathrm{depth}(u)}$. Hence the ranking structure stored for u compares them immediately. Assume from now on that the current origins are different.

The two origin depths determine two leaves of $\mathcal{D}$. Let $x \in \mathcal{D}$ be the lowest common ancestor of these two leaves, and let y_p and y_q be the children of x containing $\mathrm{depth}(\mathrm{orig}(p))$ and $\mathrm{depth}(\mathrm{orig}(q))$, respectively; see Figure 1 for an illustration. Since the child intervals of

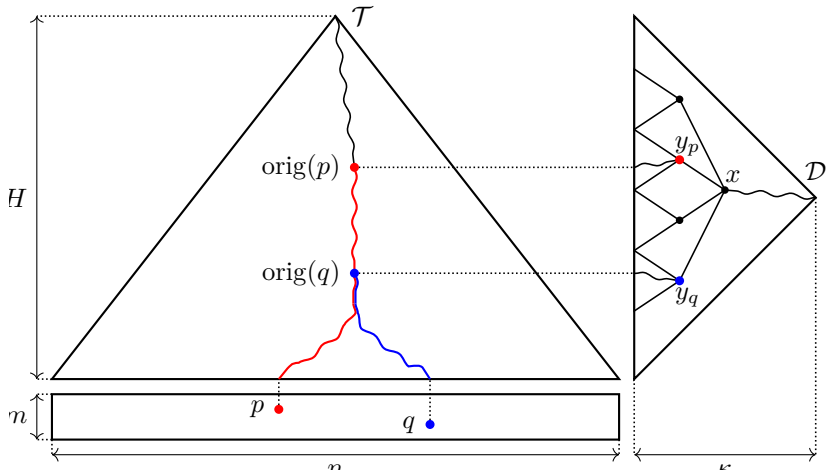

Figure 1 The query algorithm. The red and blue paths in $\mathcal{T}$ mark the nodes in which p and q are active, respectively. The depths of the two origins correspond to two leaves of $\mathcal{D}$, whose lowest common ancestor is x; the children y_p and y_q of x contain these two depths. For clarity, $\mathcal{D}$ is drawn with its root on the right, so that the left-to-right order of the children of x appears top-to-bottom.

x are consecutive depth ranges, the origin that lies in the rightmost of these two children is the deeper one. We stress that x remains fixed for the rest of the query: the target sets $T_{y,u}$ are defined so that the origin depths of the two current points never leave $\mathrm{span}(x)$, and hence the node x never needs to be recomputed.

Suppose first that $\mathrm{depth}(\mathrm{orig}(q)) > \mathrm{depth}(\mathrm{orig}(p))$. Equivalently, y_q lies strictly to the right of y_p. Let $u \coloneqq \mathrm{orig}(q)$. By Lemma 12, we have $p \in A_u$, and since $\mathrm{depth}(\mathrm{orig}(p))$ lies in a child interval of x strictly to the left of y_q, by the definition of $T_{y_q,u}$, we have $p \in T_{y_q,u}$.

We now use the lifting structure stored for (y_q, u) to obtain q', the predecessor of q in $T_{y_q,u}$. Since $p \in T_{y_q,u}$, Observation 15 tells us that $w(p) \le w(q')$ is equivalent to $w(p) \le w(q)$, and hence to $\mathcal{H}$. At this point, three cases may occur.

- **(i)** If $q' = \bot$, then the current hypothesis is false, and the query is resolved immediately.
- **(ii)** If $q' \neq \bot$ and the child $y_{q'}$ of x containing $\mathrm{depth}(\mathrm{orig}(q'))$ is different from y_p, then we replace q with q' and continue. Since $p, q' \in A_u$, the new pair (p, q') is again co-active.
- **(iii)** If $q' \neq \bot$ and $y_{q'} = y_p$, then this is the last lifting step, and it remains to compare p and q'. Let $\mathrm{span}(y_p) = [a\,..\,b]$. Since q' was obtained from the lifting structure stored at $u = \mathrm{orig}(q)$, both p and q' are active in u. Moreover, $\mathrm{depth}(u) \in \mathrm{span}(y_q)$ and y_q lies strictly to the right of y_p, so $\mathrm{depth}(u) > b$. Let v be the ancestor of u at depth b. By Lemma 9, both p and q' are active in v.
  The origin depth of p belongs to $\mathrm{span}(y_p)$ by definition of y_p, and the origin depth of q' belongs to $\mathrm{span}(y_p)$ because $y_{q'} = y_p$. Since $\mathrm{span}(y_p)$ is a canonical interval ending at b, we have $\mathrm{span}(y_p) \subseteq C_b$. Hence both points belong to L_v, and the ranking structure stored for v compares them in constant time.

The case in which $\mathrm{depth}(\mathrm{orig}(p)) > \mathrm{depth}(\mathrm{orig}(q))$, or equivalently, when y_p lies strictly to the right of y_q, is symmetric: we take $u \coloneqq \mathrm{orig}(p)$, and use the lifting structure stored for (y_p, u) to replace p with its *successor* p' in $T_{y_p,u}$, where $q \in T_{y_p,u}$ by a symmetric argument. As before, Observation 15 preserves the invariant, and the three possible outcomes mirror those of the previous case. If $p' = \top$, the hypothesis is false and the query is resolved immediately. Otherwise, if the child $y_{p'}$ of x containing $\mathrm{depth}(\mathrm{orig}(p'))$ differs from y_q, we

continue with the new co-active pair (p', q). Finally, if $y_{p'} = y_q$ and $\text{span}(y_q) = [a\,..\,b]$, we use the ranking structure stored for the ancestor of u at depth b and compare p' and q in constant time.

Each nonterminal lifting step preserves the truth of the current hypothesis and strictly decreases the child index of the deeper of the two current origins among the children of the fixed node $x \in \mathcal{D}$ (even though which of the two origins is deeper may alternate between steps). Since x has at most τ children, at most $\tau - 1$ such steps are possible before the algorithm terminates. Every step takes constant time, so the total query time is $\mathcal{O}(\tau) = \mathcal{O}(\log^{1/\kappa} n)$.

Space analysis. As in the baseline structure, the bulk of the argument relies on Lemma 17. The new complication is that the families of sets are no longer globally pairwise disjoint. To recover disjointness, we partition the stored structures into $\mathcal{O}(\kappa)$ groups, one per level of $\mathcal{D}$, so that within each group the relevant sets are pairwise disjoint. Lemma 17 then applies separately on each group, and summing over the $\mathcal{O}(\kappa)$ levels of $\mathcal{D}$ yields the claimed bound.

We begin with the local ranking structures. For every node $u \in \mathcal{T}$, the ranking structure stored for u is built for the set L_u, and its per-node cost, given by Lemma 14, coincides with the one assumed in Lemma 17 (with $X_u = L_u$). Recall that L_u consists of the points in A_u whose origin depth falls in C_d, the largest canonical interval of $\mathcal{D}$ ending at $d = \text{depth}(u)$.

We group the stored ranking structures by the level of $\mathcal{D}$ at which C_d appears, that is, by the level of the unique $\mathcal{D}$-node whose span is C_d. Fix one level ℓ, and consider all nodes $u \in \mathcal{T}$ for which $C_{\text{depth}(u)}$ appears at level ℓ. We claim that the corresponding sets L_u are pairwise disjoint. Indeed, suppose that a point p belonged to both L_u and L_v. Then $\text{depth}(\text{orig}(p))$ would lie in $C_{\text{depth}(u)} \cap C_{\text{depth}(v)}$, and since canonical intervals appearing at the same level of $\mathcal{D}$ are pairwise disjoint, this forces $C_{\text{depth}(u)} = C_{\text{depth}(v)}$, and in turn $\text{depth}(u) = \text{depth}(v)$ because each C_d ends at d. Now p is active in both u and v, with the same depth in $\mathcal{T}$, so Lemma 9 gives $u = v$, proving the claim.

Lemma 17 therefore applies and bounds the total space of the ranking structures associated with level ℓ by $\mathcal{O}(mn(\log m + \log\log n))$ bits. Summing over the $\mathcal{O}(\kappa)$ levels of $\mathcal{D}$, all ranking structures together use $\mathcal{O}(\kappa mn(\log m + \log\log n))$ bits.

We now turn to the lifting structures. Recall that for every non-leftmost child y of a node $x \in \mathcal{D}$, and every node $u \in \mathcal{T}$ with $\text{depth}(u) \in \text{span}(y)$, we store one lifting structure with source set S_u and target set $T_{y,u}$. The per-node space bound of Lemma 16 depends only on the source set and coincides with the one assumed in Lemma 17 (with $X_u = S_u$).

Here the situation differs from the ranking case. The source sets $\{S_u : u \in \mathcal{T}\}$ already partition the set of all points; what prevents a direct application of Lemma 17 is that each S_u is reused as the source set of several stored lifting structures. We claim that each S_u appears as a source set at most κ times. Indeed, S_u is the source set of the structure stored for a pair (y, u) only when y is a node of $\mathcal{D}$ with $\text{depth}(u) \in \text{span}(y)$, and there are κ such nodes, as each is an ancestor of the leaf corresponding to $\text{depth}(u)$.

We group the stored lifting structures by the level of $\mathcal{D}$ containing y. By the observation above, each S_u appears as the source set of at most one structure per group, so the source sets within a group are pairwise disjoint. Lemma 17 therefore bounds the total space of the group by $\mathcal{O}(mn(\log m + \log\log n))$ bits. Summing over the $\mathcal{O}(\kappa)$ levels of $\mathcal{D}$, all lifting structures together use $\mathcal{O}(\kappa mn(\log m + \log\log n))$ bits.

As in the baseline structure, storing $\text{depth}(\text{orig}(p))$ for every point uses $\mathcal{O}(mn \log\log n)$ bits, and the reduction structure of Lemma 7 uses $\mathcal{O}(mn \log m)$ bits. Hence the entire structure takes $\mathcal{O}(\kappa mn(\log m + \log\log n))$ bits, as promised. This concludes the proof of Theorem 1.

6 Construction of Local Primitives

We now describe the construction of the two local primitives stated in Lemmas 14 and 16. Both constructions are entirely local to one fixed node of $\mathcal{T}$, and rely on a common decomposition of the active set into at most $4m$ "monotone" subsets, called *quarter-rows*. They also make use of the two succinct encodings for sparse sets and monotone sequences introduced in Lemmas 3 and 4. The lifting structure uses the ranking structure as one of its sub-components, so we present the ranking primitive first and then build the lifting primitive on top of it.

Local coordinates. Throughout this section we fix one node $u \in \mathcal{T}$ and denote

$$\mathrm{span}(u) = [a\,..\,b] \qquad \text{and} \qquad \lambda := |\,\mathrm{span}(u)| = b - a + 1.$$

For every point $p \in [1\,..\,m] \times \mathrm{span}(u)$, we define its *local column in* u by

$$\mathrm{col}_u(p) := \mathrm{col}(p) - a + 1 \in [1\,..\,\lambda].$$

From this point onward, we will view columns through their local coordinates inside $\mathrm{span}(u)$.

We assume that the 1D RMQ structures of Lemma 2 have been built for all rows, and we will use them only to test, in constant time, whether a point is visible from a given column.

Quarter-rows. We describe a canonical decomposition of A_u into *quarter-rows* that underlies both local structures. If u is a leaf, $A_u = [1\,..\,m] \times \{j\}$ for a single column j, and we define the m quarter-rows to be the singletons $\{(i, j)\}$ for $i \in [1\,..\,m]$; all claims below are immediate in this case, so assume u is internal, with children covering the intervals $[a_1\,..\,b_1]$ and $[a_2\,..\,b_2]$.

For any $p \in A_u$, the column $\mathrm{col}(p)$ belongs to exactly one of these two child intervals. Since p is active in u, it is visible from some column in $\partial u = \{a_1, b_1, a_2, b_2\}$, and by monotonicity of visibility along its row, p is also visible from at least one endpoint of the child interval that contains $\mathrm{col}(p)$. We assign p a *canonical endpoint*: the left endpoint of the child interval containing $\mathrm{col}(p)$ if p is visible from it, and its right endpoint otherwise. For each pair $(i, c) \in [1\,..\,m] \times \{a_1, b_1, a_2, b_2\}$, the corresponding *quarter-row* is the set of all points in A_u in row i whose canonical endpoint is c; we write $Q_u(p)$ for the quarter-row containing p.

This partitions A_u into at most $4m$ quarter-rows. Given p, the quarter-row $Q_u(p)$ can be identified using $\mathcal{O}(\log m)$ bits in $\mathcal{O}(1)$ time by locating the child interval containing $\mathrm{col}(p)$ and testing visibility from its endpoints using Lemma 2.

▶ **Observation 18.** *Let Q be a quarter-row, and list its points in increasing order of weight. Then their local columns form a strictly monotone sequence. More precisely:*

- **(i)** *if Q is defined by the left endpoint of a child interval, the sequence is strictly increasing;*
- **(ii)** *if Q is defined by the right endpoint of a child interval, the sequence is strictly decreasing.*

Proof. The leaf case is trivial, so assume that u is internal. First suppose that Q is defined by the left endpoint c of one of the two child intervals of u. Let $p, q \in Q$ with $\mathrm{col}(p) < \mathrm{col}(q)$. Since both points lie in the same row and q is visible from c, the point q has the largest weight on the interval of that row between c and $\mathrm{col}(q)$. This interval contains $\mathrm{col}(p)$, so $w(p) < w(q)$. Hence the weight is a strictly increasing function of the column.

Now suppose that Q is defined by the right endpoint c of a child interval. Let again $p, q \in Q$ with $\mathrm{col}(p) < \mathrm{col}(q)$. Since p is visible from c, the point p has the largest weight on the interval of that row between $\mathrm{col}(p)$ and c. This interval contains $\mathrm{col}(q)$, so $w(p) > w(q)$. Hence the weight is a strictly decreasing function of the column. ◀

▶ **Lemma 19.** *Let Q be a quarter-row and let $X \subseteq Q$. There is an encoding using*

$$\mathcal{O}\left(|X| + |X| \log \frac{\lambda}{|X|}\right)$$

bits that, given a point $p \in X$, returns in $\mathcal{O}(1)$ time the rank of p in X.

Proof. Store the set $C_X \coloneqq \{\mathrm{col}_u(p) : p \in X\} \subseteq [1\,..\,\lambda]$ using Lemma 3. This takes $\mathcal{O}(|X| + |X| \log(\lambda/|X|))$ bits and supports rank queries on local columns in constant time.

By Observation 18, inside Q the weight is a strictly monotone function of the column. Hence, once we know the rank of $\mathrm{col}_u(p)$ in the sorted set C_X, we also know the rank of p in X. More explicitly, if the quarter-row is of the increasing type, then the two ranks coincide, whereas in the decreasing case they sum to $|X| + 1$. In both cases the answer is obtained in $\mathcal{O}(1)$ time. ◀

We are now ready to prove the two local primitives of Lemmas 14 and 16, starting with Lemma 14, which we restate below.

▶ **Lemma 14.** *Let u be a node of $\mathcal{T}$ with $\lambda_u \coloneqq |\operatorname{span}(u)|$. For every nonempty set $L \subseteq A_u$, there exists an encoding using*

$$\mathcal{O}\left(|L| \log m + |L| \log \frac{m\lambda_u}{|L|}\right)$$

bits that, given a point $p \in L$, returns in $\mathcal{O}(1)$ time the rank of p in L.

Proof. We partition L according to the quarter-rows of u. For every quarter-row Q, we define $L_Q \coloneqq L \cap Q$. Let $\mathcal{Q}$ denote the family of quarter-rows Q with $L_Q \neq \emptyset$.

The structure has two layers. The first layer handles the order inside one quarter-row, and the second layer handles how the quarter-rows are interleaved in the global order of L.

First layer. For every nonempty L_Q, we build the ranking structure of Lemma 19. Since the sets L_Q partition L, Lemma 5 with $t = |\mathcal{Q}| \leq 4m$ bounds their total space by

$$\sum_{Q \in \mathcal{Q}} \mathcal{O}\left(|L_Q| + |L_Q| \log \frac{\lambda}{|L_Q|}\right) \leq \mathcal{O}\left(|L| + |L| \log \frac{m\lambda}{|L|}\right).$$

Second layer. Let $p_1, p_2, \ldots, p_{|L|}$ be the points of L listed in increasing order of weight. For every $Q \in \mathcal{Q}$, let $B_Q \coloneqq \{i \in [1\,..\,|L|] : p_i \in L_Q\}$ be the set of positions occupied by points of L_Q in this sequence. We encode each B_Q using Lemma 3, supporting select in $\mathcal{O}(1)$ time. Since the sets L_Q partition L, Lemma 5 with $t = |\mathcal{Q}| \leq 4m$ bounds their total space by

$$\sum_{Q \in \mathcal{Q}} \mathcal{O}\left(|L_Q| + |L_Q| \log \frac{|L|}{|L_Q|}\right) \leq \mathcal{O}(|L| + |L| \log m) = \mathcal{O}(|L| \log m).$$

Query algorithm. Let $p \in L$ be the query point. We first determine its quarter-row $Q \coloneqq Q_u(p)$ in $\mathcal{O}(1)$ time. Using the structure stored for L_Q, we compute in $\mathcal{O}(1)$ time the rank r of p in L_Q. Since p is the r-th lightest point of L_Q, its rank in L equals $\mathrm{select}_{B_Q}(r)$, which we compute in $\mathcal{O}(1)$ time. ◀

We now proceed with the proof of Lemma 16, restated below.

▶ **Lemma 16.** *Let u be a node of $\mathcal{T}$ with $\lambda_u := |\operatorname{span}(u)|$. For every nonempty source set $S \subseteq A_u$ and target set $T \subseteq A_u$, there exists an encoding using*

$$\mathcal{O}\left(|S| \log m + |S| \log \frac{m\lambda_u}{|S|}\right)$$

bits that, given a point $p \in S$, returns in $\mathcal{O}(1)$ time the predecessor and successor of p in T.

Proof. We describe the structure for *successor* queries; the predecessor case is symmetric. The construction has two layers: the first determines which quarter-row contains the successor of the query point, and the second locates the successor within that quarter-row.

First layer. We build a local ranking structure of Lemma 14 for the set S, using $\mathcal{O}(|S| \log m + |S| \log(m\lambda/|S|))$ bits. We also store an array of length $|S|$, where the i-th entry is either the identifier of the quarter-row containing the successor of s_i in T, or $\top$ if no such successor exists, where $s_1, \ldots, s_{|S|}$ are the points of S listed in increasing order of weight. Since each identifier uses $\mathcal{O}(\log m)$ bits, the array uses $\mathcal{O}(|S| \log m)$ bits in total.

Second layer. For every quarter-row Q, we define

$$S_Q := \{p \in S : \text{the successor of } p \text{ in } T \text{ belongs to } Q\}.$$

Let $\mathcal{Q}$ be the family of quarter-rows Q with $S_Q \neq \emptyset$. For every $Q \in \mathcal{Q}$, we store two objects.

First, we build the ranking structure of Lemma 14 for the set S_Q. All these structures use

$$\sum_{Q \in \mathcal{Q}} \mathcal{O}\left(|S_Q| \log m + |S_Q| \log \frac{m\lambda}{|S_Q|}\right)$$

bits of space in total. The first summand contributes $\sum_{Q \in \mathcal{Q}} \mathcal{O}(|S_Q| \log m) \leq \mathcal{O}(|S| \log m)$, since the sets S_Q are pairwise disjoint subsets of S. For the second, Lemma 5 gives

$$\sum_{Q \in \mathcal{Q}} \mathcal{O}\left(|S_Q| \log \frac{m\lambda}{|S_Q|}\right) \leq \mathcal{O}\left(|S| \log \frac{m^2\lambda}{|S|}\right) = \mathcal{O}\left(|S| \log m + |S| \log \frac{m\lambda}{|S|}\right).$$

Second, let $p_1, \ldots, p_{|S_Q|}$ be the points of S_Q in increasing order of weight, and let q_i be the successor of p_i in T. We store the sequence $c_Q[i] := \operatorname{col}_u(q_i)$ of their local columns.

The sequence c_Q is monotone: if p_i is lighter than p_j, then q_i is no heavier than q_j, and since all q_i belong to the same quarter-row Q, their local columns form a monotone sequence, by Observation 18. Thus, by Lemma 4, each c_Q can be stored in $\mathcal{O}(|S_Q| + |S_Q| \log(\lambda/|S_Q|))$ bits with $\mathcal{O}(1)$-time access, and Lemma 5 bounds their total space by

$$\sum_{Q \in \mathcal{Q}} \mathcal{O}\left(|S_Q| + |S_Q| \log \frac{\lambda}{|S_Q|}\right) \leq \mathcal{O}\left(|S| + |S| \log \frac{m\lambda}{|S|}\right).$$

Query algorithm. Let $p \in S$ be the query point. We compute the rank r of p in S and inspect the r-th entry of the array stored in the first layer. If this entry is $\top$, then p has no successor in T. Otherwise, it identifies a quarter-row Q containing the successor of p. Since $p \in S_Q$, we query the local ranking structure for S_Q to obtain the rank r_Q of p in S_Q, and then extract the local column $c_Q[r_Q]$ of the successor. Since the identifier of Q determines the row of the successor, we reconstruct the successor point in constant time. ◀

References

1. Amihood Amir, Johannes Fischer, and Moshe Lewenstein. Two-dimensional range minimum queries. In Bin Ma and Kaizhong Zhang, editors, *Combinatorial Pattern Matching, 18th Annual Symposium, CPM 2007, London, Canada, July 9-11, 2007, Proceedings*, Lecture Notes in Computer Science, pages 286–294. Springer, 2007. doi:10.1007/978-3-540-73437-6_29.
2. Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. Lrm-trees: Compressed indices, adaptive sorting, and compressed permutations. *Theor. Comput. Sci.*, 459:26–41, 2012. doi:10.1016/J.TCS.2012.08.010.
3. Djamal Belazzougui, Travis Gagie, J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Range majorities and minorities in arrays. *Algorithmica*, 83(6):1707–1733, 2021. doi:10.1007/S00453-021-00799-7.
4. Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, Lecture Notes in Computer Science, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.
5. Prosenjit Bose, Evangelos Kranakis, Pat Morin, and Yihui Tang. Approximate range mode and range median queries. In Volker Diekert and Bruno Durand, editors, *STACS 2005, 22nd Annual Symposium on Theoretical Aspects of Computer Science, Stuttgart, Germany, February 24-26, 2005, Proceedings*, Lecture Notes in Computer Science, pages 377–388. Springer, 2005. doi:10.1007/978-3-540-31856-9_31.
6. Gerth Stølting Brodal, Andrej Brodnik, and Pooya Davoodi. The encoding complexity of two dimensional range minimum data structures. In Hans L. Bodlaender and Giuseppe F. Italiano, editors, *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, Lecture Notes in Computer Science, pages 229–240. Springer, 2013. doi:10.1007/978-3-642-40450-4_20.
7. Gerth Stølting Brodal, Pooya Davoodi, Moshe Lewenstein, Rajeev Raman, and Srinivasa Rao Satti. Two dimensional range minimum queries and fibonacci lattices. *Theor. Comput. Sci.*, 638:33–43, 2016. doi:10.1016/J.TCS.2016.02.016.
8. Gerth Stølting Brodal, Pooya Davoodi, and S. Srinivasa Rao. On space efficient two dimensional range minimum data structures. *Algorithmica*, 63(4):815–830, 2012. doi:10.1007/S00453-011-9499-0.
9. Timothy M. Chan, Stephane Durocher, Kasper Green Larsen, Jason Morrison, and Bryan T. Wilkinson. Linear-space data structures for range mode query in arrays. *Theory Comput. Syst.*, 55(4):719–741, 2014. doi:10.1007/S00224-013-9455-2.
10. Timothy M. Chan, Stephane Durocher, Matthew Skala, and Bryan T. Wilkinson. Linear-space data structures for range minority query in arrays. *Algorithmica*, 72(4):901–913, 2015. doi:10.1007/S00453-014-9881-9.
11. Pooya Davoodi, Gonzalo Navarro, Rajeev Raman, and S. Srinivasa Rao. Encoding range minima and range top-2 queries. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 372(2016):20130131, 2014. doi:10.1098/rsta.2013.0131.
12. Pooya Davoodi, Rajeev Raman, and Srinivasa Rao Satti. On succinct representations of binary trees. *Math. Comput. Sci.*, 11(2):177–189, 2017. doi:10.1007/S11786-017-0294-4.
13. Erik D. Demaine, Gad M. Landau, and Oren Weimann. On cartesian trees and range minimum queries. *Algorithmica*, 68(3):610–625, 2014. doi:10.1007/S00453-012-9683-X.
14. Stephane Durocher, Meng He, J. Ian Munro, Patrick K. Nicholson, and Matthew Skala. Range majority in constant time and linear space. *Inf. Comput.*, 222:169–179, 2013. doi:10.1016/J.IC.2012.10.011.
15. Hicham El-Zein, Meng He, J. Ian Munro, Yakov Nekrich, and Bryce Sandlund. On approximate range mode and range selection. In Pinyan Lu and Guochuan Zhang, editors, *30th International Symposium on Algorithms and Computation, ISAAC 2019, Shanghai University of Finance and Economics, Shanghai, China, December 8-11, 2019*, LIPIcs, pages 57:1–57:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ISAAC.2019.57.

16 Paolo Ferragina and Filippo Lari. Compressibility measures and succinct data structures for piecewise linear approximations. In Ho-Lin Chen, Wing-Kai Hon, and Meng-Tsung Tsai, editors, *36th International Symposium on Algorithms and Computation, ISAAC 2025, Tainan, Taiwan, December 7-10, 2025*, LIPIcs, pages 31:1–31:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ISAAC.2025.31.

17 Johannes Fischer and Volker Heun. Finding range minima in the middle: Approximations and applications. *Math. Comput. Sci.*, 3(1):17–30, 2010. doi:10.1007/S11786-009-0007-8.

18 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.

19 Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. An(other) entropy-bounded compressed suffix tree. In Paolo Ferragina and Gad M. Landau, editors, *Combinatorial Pattern Matching, 19th Annual Symposium, CPM 2008, Pisa, Italy, June 18-20, 2008, Proceedings*, Lecture Notes in Computer Science, pages 152–165. Springer, 2008. doi:10.1007/978-3-540-69068-9_16.

20 Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and related techniques for geometry problems. In Richard A. DeMillo, editor, *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA*, pages 135–143. ACM, 1984. doi:10.1145/800057.808675.

21 Travis Gagie, Meng He, J. Ian Munro, and Patrick K. Nicholson. Finding frequent elements in compressed 2d arrays and strings. In Roberto Grossi, Fabrizio Sebastiani, and Fabrizio Silvestri, editors, *String Processing and Information Retrieval, 18th International Symposium, SPIRE 2011, Pisa, Italy, October 17-21, 2011. Proceedings*, Lecture Notes in Computer Science, pages 295–300. Springer, 2011. doi:10.1007/978-3-642-24583-1_29.

22 Travis Gagie, Meng He, and Gonzalo Navarro. Compressed dynamic range majority and minority data structures. *Algorithmica*, 82(7):2063–2086, 2020. doi:10.1007/S00453-020-00687-6.

23 Pawel Gawrychowski, Seungbum Jo, Shay Mozes, and Oren Weimann. Compressed range minimum queries. *Theor. Comput. Sci.*, 812:39–48, 2020. doi:10.1016/J.TCS.2019.07.002.

24 Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Submatrix maximum queries in monge and partial monge matrices are equivalent to predecessor search. *ACM Trans. Algorithms*, 16(2):16:1–16:24, 2020. doi:10.1145/3381416.

25 Pawel Gawrychowski and Patrick K. Nicholson. Optimal encodings for range top- k k , selection, and min-max. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, Lecture Notes in Computer Science, pages 593–604. Springer, 2015. doi:10.1007/978-3-662-47672-7_48.

26 Pawel Gawrychowski and Patrick K. Nicholson. Optimal query time for encoding range majority. In Faith Ellen, Antonina Kolokolova, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures - 15th International Symposium, WADS 2017, St. John's, NL, Canada, July 31 - August 2, 2017, Proceedings*, Lecture Notes in Computer Science, pages 409–420. Springer, 2017. doi:10.1007/978-3-319-62127-2_35.

27 Mordecai J. Golin, John Iacono, Danny Krizanc, Rajeev Raman, Srinivasa Rao Satti, and Sunil M. Shende. Encoding 2d range maximum queries. *Theor. Comput. Sci.*, 609:316–327, 2016. doi:10.1016/J.TCS.2015.10.012.

28 Mark Greve, Allan Grønlund Jørgensen, Kasper Dalgaard Larsen, and Jakob Truelsen. Cell probe lower bounds and approximations for range mode. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I*, Lecture Notes in Computer Science, pages 605–616. Springer, 2010. doi:10.1007/978-3-642-14165-2_51.

29 Roberto Grossi, John Iacono, Gonzalo Navarro, Rajeev Raman, and S. Srinivasa Rao. Asymptotically optimal encodings of range data structures for selection and top-k queries. *ACM Trans. Algorithms*, 13(2):28:1–28:31, 2017. doi:10.1145/3012939.

30 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

31 Seungbum Jo and Geunho Kim. Space-efficient data structure for next/previous larger/smaller value queries. *Algorithmica*, 87(10):1369–1392, 2025. doi:10.1007/S00453-025-01325-9.

32 Seungbum Jo, Rahul Lingala, and Srinivasa Rao Satti. Encoding two-dimensional range top-k queries. *Algorithmica*, 83(11):3379–3402, 2021. doi:10.1007/S00453-021-00856-1.

33 Seungbum Jo and Srinivasa Rao Satti. Simultaneous encodings for range and next/previous larger/smaller value queries. *Theor. Comput. Sci.*, 654:80–91, 2016. doi:10.1016/J.TCS.2016.01.043.

34 Seungbum Jo and Srinivasa Rao Satti. Encoding data structures for range queries on arrays. In Alessio Conte, Andrea Marino, Giovanna Rosone, and Jeffrey Scott Vitter, editors, *From Strings to Graphs, and Back Again: A Festschrift for Roberto Grossi's 60th Birthday, Grossi's Festschrift, Venice, Italy, July 25, 2025*, OASIcs, pages 12:1–12:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/OASIcs.GROSSI.12.

35 Seungbum Jo and Srinivasa Rao Satti. Encodings for range minimum queries over bounded alphabets. *Theor. Comput. Sci.*, 1070:115824, 2026. doi:10.1016/J.TCS.2026.115824.

36 Haim Kaplan, Shay Mozes, Yahav Nussbaum, and Micha Sharir. Submatrix maximum queries in monge matrices and partial monge matrices, and their applications. *ACM Trans. Algorithms*, 13(2):26:1–26:42, 2017. doi:10.1145/3039873.

37 Marek Karpinski and Yakov Nekrich. Searching for frequent colors in rectangles. In *Proceedings of the 20th Annual Canadian Conference on Computational Geometry, Montréal, Canada, August 13-15, 2008*, 2008.

38 Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *Nord. J. Comput.*, 12(1):1–17, 2005.

39 Mingmou Liu. Nearly tight lower bounds for succinct range minimum query. *CoRR*, abs/2111.02318, 2021. arXiv:2111.02318.

40 Mingmou Liu and Huacheng Yu. Lower bound for succinct range minimum query. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 1402–1415. ACM, 2020. doi:10.1145/3357713.3384260.

41 J. Ian Munro, Patrick K. Nicholson, Louisa Seelbach Benkner, and Sebastian Wild. Hypersuccinct trees - new universal tree source codes for optimal compressed tree data structures and range minima. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, Lisbon, Portugal (Virtual Conference), September 6-8, 2021*, LIPIcs, pages 70:1–70:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ESA.2021.70.

42 Gonzalo Navarro and Sharma V. Thankachan. Optimal encodings for range majority queries. *Algorithmica*, 74(3):1082–1098, 2016. doi:10.1007/S00453-015-9987-8.

43 Holger Petersen. Improved bounds for range mode and range median queries. In Viliam Geffert, Juhani Karhumäki, Alberto Bertoni, Bart Preneel, Pavol Návrat, and Mária Bieliková, editors, *SOFSEM 2008: Theory and Practice of Computer Science, 34th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 19-25, 2008, Proceedings*, Lecture Notes in Computer Science, pages 418–423. Springer, 2008. doi:10.1007/978-3-540-77566-9_36.

44 Holger Petersen and Szymon Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Inf. Process. Lett.*, 109(4):225–228, 2009. doi:10.1016/J.IPL.2008.10.007.

45 Chung Keung Poon. Optimal range max datacube for fixed dimensions. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, Lecture Notes in Computer Science, pages 158–172. Springer, 2003. doi:10.1007/3-540-36285-1_11.

46 Rajeev Raman. Encoding data structures. In M. Sohel Rahman and Etsuji Tomita, editors, *WALCOM: Algorithms and Computation - 9th International Workshop, WALCOM 2015, Dhaka, Bangladesh, February 26-28, 2015. Proceedings*, Lecture Notes in Computer Science, pages 1–7. Springer, 2015. doi:10.1007/978-3-319-15612-5_1.

47 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

48 Kunihiko Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007. doi:10.1016/J.JDA.2006.03.011.

49 Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. In John H. Reif, editor, *VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing, AWOC 88, Corfu, Greece, June 28 - July 1, 1988, Proceedings*, Lecture Notes in Computer Science, pages 111–123. Springer, 1988. doi:10.1007/BFB0040379.

50 Matthew Skala. Array range queries. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, Lecture Notes in Computer Science, pages 333–350. Springer, 2013. doi:10.1007/978-3-642-40273-9_21.

51 Dekel Tsur. The effective entropy of next/previous larger/smaller value queries. *Inf. Process. Lett.*, 145:39–43, 2019. doi:10.1016/J.IPL.2019.01.011.

52 Hao Yuan and Mikhail J. Atallah. Data structures for range minimum queries in multidimensional arrays. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 150–160. SIAM, 2010. doi:10.1137/1.9781611973075.14.

A Omitted proofs

▶ **Lemma 5** (log-sum inequality). *For any positive numbers $k_1, \ldots, k_t$ and $U_1, \ldots, U_t$, with total sums $K \coloneqq \sum_{i=1}^{t} k_i$ and $U \coloneqq \sum_{i=1}^{t} U_i$,*

$$\sum_{i=1}^{t} k_i \log \frac{U_i}{k_i} \leq K \log \frac{U}{K}.$$

Proof. For every i, let $\alpha_i \coloneqq k_i/K$, so that $\sum_{i=1}^{t} \alpha_i = 1$. Then

$$\sum_{i=1}^{t} k_i \log \frac{U_i}{k_i} = K \sum_{i=1}^{t} \alpha_i \log \frac{U_i/K}{\alpha_i}.$$

Since $\sum_{i=1}^{t} \alpha_i = 1$ and the logarithm is concave, Jensen's inequality yields

$$\sum_{i=1}^{t} \alpha_i \log \frac{U_i/K}{\alpha_i} \leq \log\left(\sum_{i=1}^{t} \alpha_i \cdot \frac{U_i/K}{\alpha_i}\right) = \log\left(\sum_{i=1}^{t} \frac{U_i}{K}\right) = \log \frac{U}{K}.$$

Multiplying both sides by K gives the claim. ◀