Data Structures for Range Minimum Queries in Multidimensional Arrays [*]

Hao Yuan [†]   Mikhail J. Atallah [‡]

Abstract

Given a d-dimensional array A with N entries, the *Range Minimum Query (RMQ)* asks for the minimum element within a contiguous subarray of A. The 1D RMQ problem has been studied intensively because of its relevance to the *Nearest Common Ancestor* problem and its important use in stringology. If constant-time query answering is required, linear time and space preprocessing algorithms were known for the 1D case, but not for the higher dimensional cases. In this paper, we give the first linear-time preprocessing algorithm for arrays with fixed dimension, such that any range minimum query can be answered in constant time.

1   Introduction

Problem Definition. Given is a d-dimensional array A of size $N = n_1 \cdot n_2 \cdot \cdots \cdot n_d$, where n_k $(1 \leq k \leq d)$ is the length of the k_{th} dimension. The entries in A are from a linearly ordered set S (under the relation $\leq$). A *d-dimensional Range Minimum Query (d-RMQ)* asks the minimum element in the query range $q = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$, i.e.,

$$\mathrm{RMQ}(A, q) = \min A[q] = \min_{(i_1, \ldots, i_d) \in q} A(i_1, \ldots, i_d).$$

The goal of the d-RMQ problem is to preprocess the input array A efficiently (in terms of time and space), so that any online range minimum query can be answered efficiently.

Through out this paper we assume that: an element in S can only involve in comparisons with other elements in S, even in the RAM model implementation (i.e., we do not assume they are integers); the comparison of two S elements takes constant time.

Previous Work. The 1D RMQ problem has been studied intensively because of its important use in many algorithms, especially those in stringology [12]. A linear time and space preprocessing algorithm for 1D RMQ to achieve constant-time query answering was first given by Gabow, Bentley and Tarjan [10], using a linear reduction to the *Nearest Common Ancestor (NCA)* problem [15] on the Cartesian tree [20]. The Cartesian tree (defined by Vuillemin [20]) is built on top of the input array to completely capture the information necessary to determine the solution for any range minimum query on the original array. It was shown that any range minimum query can be translated to an NCA query on the precomputed Cartesian tree in constant time. The first constant-time NCA solution with linear preprocessing was given by Harel and Tarjan [15], and many efforts were spent on simplifying the solution [18, 2, 4].

The RMQ problem for high dimensional points in $\mathbb{R}^d$ (not necessary in a d-dimensional array) was first studied by Gabow, Bentley and Tarjan [10]. They presented a data structure to preprocess N points in $O(N \log^{d-1} N)$ time and space, so that any range minimum query can be answered in $O(\log^{d-1} N)$ time. The technique used was the range trees introduced by Bentley [5] to recursively process multidimensional queries using dimension-reduction.

In the setting of range searching in multidimensional arrays, better solutions exist. Chazelle and Rosenberg [6] gave an algorithm to compute the range sum in the semigroup model. In their formulation, the elements in A were from a semigroup $(S, +)$ and a range query q asks the sum of the elements in $A[q]$. Because $(S, \min)$ can be viewed as a semigroup (when S is linearly ordered), Chazelle and Rosenberg's preprocessing scheme can be applied to solve the RMQ problem. Using M units of storage, in the case that d is fixed and $d \leq \log_{14} M/N$, their algorithm preprocesses an input array in $O(M)$ time, and can answer any range sum (or range min) query in $O(\alpha^d(M, N))$ time, where α is the functional inverse of Ackermann's function defined by Tarjan [19]. To make the querying time constant, the space M in their data structure needs to be superlinear (i.e., $O(N(\lambda(k, N))^d)$) for some constant k, where

$\lambda(k, N)$ is an inverse-Ackermann type function defined in [1]). Note that a unit of storage corresponds to a semigroup element.

Linear space data structures that allow constant querying time were developed by Poon [17]. In his work, data structures for range max/min queries in d-dimensional OLAP datacubes were given, where an OLAP datacube (see [7, 11]) is modeled as a multidimensional array. For a d-dimensional array with each dimension of the same length n (i.e., $N = n^d$), his data structure can be built in $O((3n \log^* n)^d)$ time using $O((3n)^d)$ words, assuming each word has $c_1 \log n$ bits for some constant c_1 and $d \leq c_2 \log \log n / \log(\log^* n)$ for some constant c_2; the data structure can answer any range minimum query in $O(8^d)$ time.

Amir, Fischer and Lewenstein [3] considered the two-dimensional case and presented a class of algorithms that can solve the 2D RMQ problem using $O(kN)$ additional space (in terms of words), $O(N \log^{[k+1]} N)$ preprocessing time and $O(1)$ querying time for any $k > 1$, where $\log^{[k+1]} N$ is the result of applying the log function $k + 1$ times on N. The key idea of their algorithm was to precompute the solutions for a small number of micro blocks, where the size of each micro block is $O(s^2)$ with $s = \log^{[k]} n$ and $n = \sqrt{N}$. Based on the fact that, if two micro blocks have the same permutation type, then the positions (relative to the corner of the micro block) of any range minimum query on those two micro blocks will be the same (assuming the elements are distinct). Since the number of permutations of size s^2 is very small (e.g., the number is less than $n^{1/10}$ when n is large enough), it is affordable to precompute the answers (relative offsets) of all range minimum queries for all permutations of size s^2. The superlinear preprocessing time of the whole algorithm comes from identifying the permutation types of the micro blocks, because each micro block needs to be identified by sorting its elements in $O(s^2 \log s)$ time, resulting in a total of $O(n^2/s^2 \cdot s^2 \log s^2) = O(N \log s)$ preprocessing time for all n^2/s^2 blocks. Note that, if elements in S are integers, the sorting can be done faster than $O(s^2 \log s)$ in the word RAM model using the integer sorting algorithms of Han [13], Han and Thorup [14].

Instead of computing the permutation types of micro blocks, a natural thought is to consider structures similar to the Cartesian trees used in the 1D case, that satisfy the following properties:

- (1) The structure must completely capture the positions of all RMQ results;

- (2) Its encoding can be computed in linear time.

Note that a permutation type has the first property, but not the second one.

In a recent result of Demaine, Landau and Weimann [8], it was shown that no such kind of Cartesian-tree-like structure exists in the 2D case. More specifically, they show that the number of different 2D RMQ $n \times n$ matrices is $\Omega\left(\left(\frac{n}{4}!\right)^{n/4}\right)$, where two matrices are considered different only if their range minima are in different locations for some rectangular range. This implies that, any encoding (or "type") for an $s \times s$ micro block that satisfies the first property, needs to be computed using at least $\log(\frac{s}{4}!)^{s/4} = \Omega(s^2 \log s)$ time (i.e., it violates the second property).

Our Contribution. We give the first linear-time preprocessing algorithm for constant-time RMQ querying in 2D (or higher dimensions). As it is provably impossible to use a Cartesian-tree-like structure to solve the 2D RMQ problem [8], we had to use a new technique that uses a new type of encoding method. The core of our new method is a structure that has the following properties:

- (1) It does not need to completely capture the positions of all RMQ results, instead, it should be able to generate a constant number of candidates at the querying stage to compare.

- (2) Its encoding can be computed in linear time.

Note that only property (1) is changed from the requirement of a Cartesian-tree-like structure. The breakthrough comes from the fact that, a Cartesian-tree-like structure requires to get the range minimum without any comparison at the querying stage, while the new structure allow some comparisons to be made in order to decide the range minimum. The flexibility to decide the range minimum at the querying stage greatly reduces the number of types to linear.

We will present our results in two steps. The first step is to consider the *comparison* complexity of the preprocessing and querying, i.e., only the number of pairwise comparisons of elements from S is counted. We will show that $B(d)N$ comparisons are sufficient to preprocess a d-dimensional array (where $B(d) = O\left((2.89)^d \cdot (d+1)!\right)$), such that only $2^d - 1$ comparisons are required to answer a range minimum query at the querying stage. The second step is to implement the scheme under the RAM model when d is fixed. The querying time is increased to $O(3^d)$ in the RAM model if $O(2^d d! N)$ words additional to the input are used, and the preprocessing time is $O(B(d)N)$.

Roadmap. This paper is organized as follows. Section 2 defines basic notations. Section 3 gives a new 1D RMQ data structure in both the comparison and RAM

models. Section 4 gives our data structures for the 2D and higher dimensional cases in the comparison model, and Section 5 discusses the RAM implementations.

2 Preliminaries

To make the presentation cleaner, throughout this paper we assume that the length of each dimension (i.e., n_k for $1 \leq k \leq d$) is a power of 2. This assumption is done without loss of generality, and dropping it would result in a more cluttered exposition but would not cause any change in our asymptotic complexity bounds.

The indices of the k_{th} dimension are assumed to be $[1, n_k]$. For a range $q = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$, we define $A[q]$ to be the subarray induced by q, and $\text{RMQ}(A, q) = \min A[q]$ to be the minimum element in $A[q]$. In an induced subarray $A[q]$, the indices of the k_{th} dimension are assumed to be $[1, b_k - a_k + 1]$.

Without loss of generality, assume that the entries in A are distinct. Under this assumption, there will be only one minimum element in any range; the assumption can be forced to hold even if A had multiple entries, by breaking ties according to the lexicographic ordering of the indices of A that contain the equal values. Define $\text{POS}(A, q)$ to be the d-dimensional index of the only minimum element in q, i.e., the d-length vector $(x, y, z, \cdots)$ of the integer coordinates of that minimum's position. For a d-dimensional index i, the notation $A[i]$ represents the element indexed by i. An explicit representation of an entry using the coordinates is $A(x, y, z, \cdots)$, e.g., $A(x, y)$ represents the element at the x_{th} row and y_{th} column of a two dimensional array A.

The size of a range (or an interval in the 1D case) is defined to be the number of integer coordinates that are contained in the range. For example, the size of $[a, b] \times [c, d]$ will be $(b - a + 1)(d - c + 1)$ where a, b, c and d are all integers. The size of a range q is denoted by $|q|$. If $|q| = 1$, then $A[q]$ represents an array entry.

3 New One-Dimensional RMQ

Most of the previous linear data structures for solving the 1D RMQ problem are based on the following fact: there exists an $O(N)$-bits structure (also built within N comparisons) such that any RMQ query can be answered without any comparison. A popular structure that has this property is the Cartesian tree [20]. Unfortunately, as shown by Demaine, Landau and Weimann [8], it is impossible to generalize the Cartesian tree to higher dimensions. This is why this section introduces a new structure for solving the 1D RMQ problem, one that generalizes to higher dimensions. Unlike using the Cartesian tree, some comparisons are required in the querying stage using our structure. For this 1D case,

we use n instead of N in what follows.

3.1 Linear Preprocessing in the Comparison Model We will first present a data structure of size $O(n \log n)$, and then show that $4n$ comparisons are sufficient to build the data structure. In the querying stage, any range minimum query will involve at most 1 comparison.

3.1.1 Preprocessing. Like the range trees in [5], we build a set of *canonical intervals* in the following way: The interval $[1..n]$ is a canonical interval. Split the interval $[1..n]$ into two sub-intervals: the left one is $[1..\frac{n}{2}]$, and the right is $[\frac{n}{2} + 1, n]$. The two sub-intervals are also canonical intervals. Recurse the splitting process on the two sub-intervals to generate more canonical intervals, until the size of the interval is 1 (see Section 2 for the definition of the size of a range).

There are 2^k canonical intervals of size $n/2^k$ for $0 \leq k \leq \log n$, so there are totally $2n - 1$ canonical intervals. For each canonical interval $I = [a, b]$, we build the following data structures: for $a \leq x \leq b$,

- $\text{LeftMin}(I, x) = \min_{i \leq x \text{ and } i \in I} A[i]$;

- $\text{RightMin}(I, x) = \min_{i \geq x \text{ and } i \in I} A[i]$.

Although there are totally $O(n \log n)$ entries to compute, we will demonstrate a dynamic programming method to compute the entries using only a linear number of comparisons [1].

Consider two neighboring canonical intervals $I_1 = [a, b]$ and $I_2 = [b + 1, c]$. Assume that the LeftMin structures were already computed for them, then the following algorithm can compute LeftMin for the interval $I = I_1 \cup I_2 = [a, c]$ in $\lceil \log_2(c - b + 1) \rceil$ comparisons.

- For $x \in I_1$, we have $\text{LeftMin}(I, x) = \text{LeftMin}(I_1, x)$. This step does not need any comparison, as it simply copies the already computed LeftMin entries from the interval I_1.

- For $x \in I_2$, observe that $[a, x] = [a, b] \cup [b + 1, x]$, implying

  $\text{LeftMin}(I, x) = \min \{\text{LeftMin}(I_1, b), \text{LeftMin}(I_2, x)\}.$

  Let $s = \text{LeftMin}(I_1, b)$. A direct method using the above observation is to compare s to $\text{LeftMin}(I_2, x)$ for each $x \in I_2$, and assign the minimum value to $\text{LeftMin}(I, x)$. This naive approach will cost

---

[1]Of course, using the Cartesian tree technique, $2n - 1$ comparisons are enough. However, our technique is for generalization to higher dimensions, where the Cartesian tree technique no longer applies.

$|I_2| = c - b$ comparisons. To reduce the number of comparisons, we use another observation that

$$[b+1, b+1] \subset [b+1, b+2] \subset \cdots \subset [b+1, c-1] \subset [b+1, c],$$

which implies $\text{LeftMin}(I_2, x_1) \geq \text{LeftMin}(I_2, x_2)$ for any $x_1 \leq x_2$. In the other words, $\text{LeftMin}(I_2, x)$ is non-increasing as x goes from $b+1$ to c. Therefore, a binary search of s in the LeftMin structure of I_2 is sufficient to build the LeftMin structure for I. The binary search costs $\lceil \log_2(|I_2|+1) \rceil = \lceil \log_2(c-b+1) \rceil$ comparisons.

Similarly, the RightMin structure for the interval I can also be built in $\lceil \log_2(|I_1| + 1) \rceil$ comparisons. Denote the above process by $\text{Merge}(I_1, I_2)$. The number of comparisons in $\text{Merge}(I_1, I_2)$ is then $\lceil \log_2(|I_1| + 1) \rceil + \lceil \log_2(|I_2| + 1) \rceil = \lceil \log_2(b - a + 2) \rceil + \lceil \log_2(c - b + 1) \rceil$.

Using this merging process, one can do a bottom-up merging as follows: Initially, the LeftMin and RightMin structures for size 1 canonical intervals can be assigned without any comparison; Assume that the structures for all size 2^k canonical intervals were computed, then the structures for any size 2^{k+1} canonical interval I can be computed by $\text{Merge}(I_1, I_2)$ where I_1 and I_2 are the two canonical intervals split directly from I. The number of comparisons spent on a size l interval is bounded by $2\lceil \log_2(l/2 + 1) \rceil \leq 2 \log_2 l$.

If we implement the above process in a top-down recursion fashion, and let $H(n)$ denote the number of comparisons used to preprocess an array of size n, then we have $H(1) = 0$ and for $n \geq 2$

$$H(n) = 2H(n/2) + 2 \log n.$$

The above recursion yields $H(n) \leq 4n$. The saving of comparisons comes from the logarithmic-comparison merging process. This $O(n/\log n)$ factor saving can be generalized to higher dimensional cases as shown in Section 4.

Although the number of comparisons used in this preprocessing algorithm is linear, a naive implementation in RAM will require $O(n \log n)$ time.

3.1.2 Basic Querying

Consider a query $q = [a, b]$, we can compute $\text{RMQ}(A, q)$ by calling $\text{SolveQuery}([1, n], q)$ below:

Procedure SolveQuery(I, q) : Assume that $I = [u, v]$.

- Case 1: If $a = u$, then return $\text{LeftMin}(I, b)$; if $b = v$, then return $\text{RightMin}(I, a)$;

- Case 2: Let I_1 and I_2 be the two canonical intervals split from I. If $q \subseteq I_1$, then return $\text{SolveQuery}(I_1, q)$. If $q \subseteq I_2$, then return $\text{SolveQuery}(I_2, q)$.

- Case 3: If neither Case 1 nor Case 2 applies, then q must overlap with both I_1 and I_2. In this case, q contains the right boundary of I_1 and the left boundary of I_2, so we have $\text{RMQ}(A, q) = \min\{\text{RightMin}(I_1, a), \text{LeftMin}(I_2, b)\}$.

For any query q, the above procedure will eventually reach Case 1 or Case 3. Case 1 makes no comparison[2], and Case 3 only needs 1 comparison. Therefore, at most 1 comparison is required for a query.

3.1.3 Faster Querying

A direct implementation of this querying algorithm in RAM requires $O(\log n)$ time. The inefficiency comes from the recursion in Case 2. To speed it up, the following scheme jumps to Case 1 or 3 in $O(1)$ time:

- **Preprocessing.** Build an auxiliary tree T based on the canonical intervals where: each canonical interval I corresponds to a tree node v_I; for any canonical interval I, let I_1 and I_2 be the two canonical intervals that are directly split from I, then make the tree node v_I the parent of v_{I_1} and v_{I_2}. Preprocess a nearest common ancestor data structure for the tree T using any of the linear-time algorithm discussed in Section 1. This preprocessing can be done in linear time and space.

- **Querying.** Assume that the query is $q = [a, b]$, then let I_1 be the canonical interval $[a, a]$ and I_2 be the canonical interval $[b, b]$. Find the nearest common ancestor of v_{I_1} and v_{I_2} in T in constant time. Denote the resulting nearest common ancestor to be v_I (associated with a canonical interval I), then we have $q \subseteq I$. It can be shown that either Case 1 or Case 3 applies if we call $\text{SolveQuery}(I, q)$ to compute the range minimum for q. Therefore, the querying time is now improved to be constant.

3.2 Efficient Implementation in RAM

A straightforward implementation of the above linear-comparison preprocessing algorithm will require $O(n \log n)$ preprocessing time and storage. To speed up the time and save space, we will use a technique of Dixon, Rauch and Tarjan [9]. Their technique was used to implement the linear-comparison MST (Minimum Spanning Tree) verification algorithm of Komlós [16] in linear time and space in the RAM model.

The basic idea (same as in other 1D RMQ algorithms) is to divide the input array into micro blocks of size $g = c_1 \log n$, where c_1 is a constant that will be

---

[2]Recall that, in the comparison model, we count only the comparisons between the elements from the linearly ordered set S (but not other comparisons, e.g., between indices).

determined later. For $1 \leq i \leq n/g$, define block B_i to be $A[b_i]$, where $b_i = [(i-1)g+1, ig]$. Consider a query q:

- Case 1. If q is a subset of any b_i, then we will apply the micro data structure that is built for B_i to answer the query. The algorithm to build the micro data structure will be covered later in Section 3.2.2.

- Case 2. If q overlaps with two micro blocks (assumed to be b_i and b_{i+1}), then the query is divided into two subqueries $q \cap b_i$ and $q \cap b_{i+1}$, to which Case 1 can apply.

- Case 3. Assume that the query overlaps with more than two micro blocks, where the micro blocks are $b_i, b_{i+1}, \cdots, b_j$. Then we divide q into three subqueries: $q \cap b_i$, $q \cap b_j$ and $q \cap \bigcup_{i<k<j} b_k$. The first two subqueries are in Case 1. The third subquery is solved by a macro data structure (see below).

3.2.1 Macro Data Structure Create an RMQ instance A' of size n/g by setting $A'[i] = \min B_i$ for $1 \leq i \leq n/g$. Then any query of the form $q \cap \bigcup_{i<k<j} b_k$ (the third subquery in Case 3) can be translated to $\text{RMQ}(A', [i+1, j-1])$. To solve the RMQ problem on A', we can apply the algorithm in Section 3.1 to A'. The preprocessing time and space (in words, not bits) are $O(|A'| \log |A'|) = O(n/g \log(n/g)) = O(n)$.

3.2.2 Micro Data Structures We define the *type* of a micro block B, denoted by $\text{Type}(B)$, in the following way: Run the linear-comparison preprocessing algorithm of Section 3.1 on B. During the execution of the algorithm, a sequence of comparisons will be made. Denote the comparison results to be $\text{Res} = \langle r_1, r_2, \cdots, r_p \rangle$, where p is the number of comparisons, and r_i is the result of the i_{th} comparison. The value of r_i is 0 if the "$\leq$" comparison returns false, and 1 otherwise. So Res is a 0/1 vector of length p. Because $p \leq 4g$, we can encode Res using $4g+1$ bits (i.e., append a bit of 1 and then $4g - p$ bits of 0's to mark the end of comparisons). This encoding is the type of B.

Let B_1 and B_2 be two micro blocks of the same type and size. For any RMQ query $q \subseteq [1, g]$, the cases encountered in the the querying algorithm (i.e., SolveQuery) of Section 3.1 will be identical. If they eventually reach Case 1 of SolveQuery, then we will have $\text{POS}(B_1, q) = \text{POS}(B_2, q)$ (see Section 2 for the definition of POS) because their types are the same. If they reach Case 3 of SolveQuery, then the indices (i.e., relative offsets) of the two to-be-compared *candidates* (one from RightMin and the other from LeftMin) are the same within both B_1 and B_2.

The above analysis yields the following preprocessing algorithm: For each possible micro block type t, run the linear-comparison preprocessing algorithm from Section 3.1 with the following modification: Instead of storing the minimum elements in the LeftMin and RightMin structure, we store the indices of the minimum elements. For example, instead of storing $\text{LeftMin}(I, x)$ for some canonical interval I and parameter x, we store $\text{PosLeftMin}(I, x)$, the index of $\text{LeftMin}(I, x)$ in the original array. For each possible query $q \subseteq [1, g]$, compute the indices of the two candidates (sometimes only one candidate) and store them at $\text{Indices}(t, q)$. Note that $\text{Indices}(t, q)$ is a list of indices of candidates, and denote $\text{Indices}(t, q)[i]$ ($0 \leq i \leq 1$) to be the index of the i_{th} candidate for query q in a micro block of type t.

To enumerate all the types and build the "Indices" structures, we can construct the following decision tree

- Assume that we have a deterministic program in the RAM model to simulate the linear-comparison preprocessing algorithm. The input to the program is just the size of the input array (i.e., g), and the program is not able to retrieve any element in the array. However, the program is allowed to access a comparison oracle to compare any two array entries given their indices.

- Each internal tree node contains a configuration (i.e., program states) of the random access machine and a pair of indices. The tree node corresponds to a request to the comparison oracle, where the indices indicate which array entries to compare. The configuration stored at the tree node is the configuration of the machine right before the program sends the comparison request. Note that, a configuration includes a full set of PosLeftMin and PosRightMin structures. Therefore, the size of a configuration is $O(g \log g)$.

- In the configuration of the root node, we have $\text{PosLeftMin}(I(x), x) = x$ and $\text{PosRightMin}(I(x), x) = x$ for each $1 \leq x \leq g$ and its corresponding canonical interval $I(x) = [x, x]$.

- Each leaf of the decision tree contains a configuration representing the halting state of the program.

- Each internal tree node has two children, representing the two branching cases according to the result returned from the comparison oracle.

- The depth of the decision tree is at most $4g$.

- A path from the root to a leaf corresponds to an execution of the program.

- For each leaf node, the encoding of the comparison results (on the path from the root to the leaf) corresponds to a micro block type.

To construct the decision tree described above, simply use a depth-first or breadth-first approach to grow the tree. Growing each tree node takes at most $O(g \log g)$ time and space. Therefore, the total time and space to construct the decision tree is $O(2^{4g} \cdot g \log g)$.

Now, we have all the possible types (corresponding to the leaves of the decision tree). For each type t, build the Indices(t, q) list for every possible query q, based on the already computed preprocessing data structures (part of the configuration) stored at the corresponding leaf. There are $O(g^2)$ possible queries for each type, and for each query it takes $O(1)$ time to find the indices of the candidates. Therefore, the total time and space to build all the Indices structures are $O(2^{4g} \cdot g^2)$. Note that the configurations stored at the tree nodes can be discarded after computing the Indices structures.

For each micro block B_i, the Type(B_i) is obtained in $O(g)$ time by walking down the decision tree according to the comparison results based on the real array instance B_i. The total time to compute the types of the n/g micro blocks will be $O(n/g \cdot g) = O(n)$.

For any micro block B and a query q, we are able to find the indices of the candidates by a constant-time table lookup from Indices$(\text{Type}(B), q)$, and compute the query answer by

$$\text{RMQ}(B, q) = \min_i B[\,\text{Indices}(\text{Type}(B), q)[i]\,].$$

The subindex i goes from 1 to the length of the list Indices$(\text{Type}(B), q)$. The querying time for any subquery in any micro block is $O(1)$ based on this scheme.

Based on the above analysis, the total preprocessing time to build the data structures for all micro blocks is $O(n + 2^{4g} \cdot g^2)$. The space complexity is $O(n/g \cdot (4g + 1) + 2^{4g} g^2 \cdot \log g) = O(n + 2^{4g} g^2 \cdot \log g)$ bits, where the term $n/g \cdot (4g + 1)$ comes from storing the types for n/g micro blocks, and $2^{4g} g^2 \cdot \log g$ comes from storing $O(2^{4g} g^2)$ Indices structure with each index using $O(\log g)$ bits. Setting $g = c_1 \log n$, we get $2^{4c_1 \log n}(c_1 \log n)^2 \log(c_1 \log n) = n^{4c_1}(c_1 \log n)^2 \log(c_1 \log n)$. Therefore, choosing c_1 to be any constant less than $1/4$ will make the whole preprocessing complexities (both time and additional bit space) to be $O(n)$.

To summarize this section: all the subqueries can be answered in constant time with the macro and micro data structures, where the data structures can be built in linear time and space.

4   Multidimensional RMQ in the Comparison Model

This section covers the comparison complexities for 2D and higher dimensional RMQ problem.

4.1   Two-Dimensional RMQ
We assume that the input array A is m by n. Therefore, we will have $N = mn$.

4.1.1   Linear-Comparison Preprocessing
Following Section 3.1, construct the canonical intervals for $[1, m]$ and $[1, n]$ respectively. Denote the canonical interval set for $[1, m]$ by CIS_1, and for $[1, n]$ by CIS_2. The set of canonical ranges for the 2D array are

$$\text{CR} = \big\{ I_1 \times I_2 \mid I_1 \in \text{CIS}_1 \text{ and } I_2 \in \text{CIS}_2 \big\}.$$

The number of canonical ranges in CR is $|\text{CIS}_1| \cdot |\text{CIS}_2| \le 4mn$, i.e., $|\text{CR}| = O(N)$.

Within each canonical range $r = [a_1, b_1] \times [a_2, b_2]$, we build the following data structures (call them *DominanceMin structures*) for each index $(x, y) \in r$,

$$\text{TopLeftMin}(r, (x, y)) = \min_{a_1 \le i \le x, a_2 \le j \le y} A(i, j);$$

$$\text{TopRightMin}(r, (x, y)) = \min_{a_1 \le i \le x, y \le j \le b_2} A(i, j);$$

$$\text{BottomLeftMin}(r, (x, y)) = \min_{x \le i \le b_1, a_2 \le j \le y} A(i, j);$$

$$\text{BottomRightMin}(r, (x, y)) = \min_{x \le i \le b_1, y \le j \le b_2} A(i, j).$$

These data structures precompute the results for any 2-side dominance-min queries within each canonical range. The total space used for the DominanceMin structures is clearly $O(mn \log m \log n)$, or loosely $O(N \log^2 N)$. A straightforward implementation will cost $O(N \log^2 N)$ comparisons.

To reduce the number of comparisons to linear, we can use the following dynamic programming approach to construct the DominanceMin structures: First, sort the canonical ranges in the increasing order of their sizes. This will take $O(|\text{CR}| \log |\text{CR}|) = O(N \log N)$ time. Then consider the sorted canonical ranges one by one: For canonical ranges of size 1, it is trivial to initialize their DominanceMin structures without any comparison. For a canonical range $r = I_1 \times I_2$, whose size is greater than 1, we can build the DominanceMin structures for it by combining the DominanceMin structures of two smaller canonical ranges with at most $4|I_2| \log |I_1|$ comparisons in the case when $|I_1| \ge |I_2|$, or $4|I_1| \log |I_2|$ comparisons in the case when $|I_1| < |I_2|$. We will only show the algorithm for the case when $|I_1| \ge |I_2|$ and only compute the TopLeftMin structures, because the other cases can be done in a similar way.

Let $I_1 = [a_1, b_1]$ and $I_2 = [a_2, b_2]$. In the case when $|I_1| \geq |I_2|$, let I_3 and I_4 be the two canonical intervals directly split from I_1, where $I_3 = [a_1, (a_1+b_1-1)/2]$ and $I_4 = [(a_1 + b_1 + 1)/2, b_1]$. We combine the TopLeftMin structures from the smaller canonical ranges $r_1 = I_3 \times I_2$ and $r_2 = I_4 \times I_2$. For any index $(x, y) \in r_1$, we have TopLeftMin$(r, (x, y)) = $ TopLeftMin$(r_1, (x, y))$. For any index $(x, y) \in r_2$, let $s = $ TopLeftMin$(r_1, ((a_1 + b_1 - 1)/2, y))$; then we have

TopLeftMin$(r, (x, y)) = \min\{s, \text{TopLeftMin}(r_2, (x, y))\}$.

Similar to our 1D scheme, we will not use the above equation directly. Observe that the sequence $\{$TopLeftMin$(r_2, (x, y))\}$ for $x \in I_4$ (i.e., fixing y) is non-increasing, therefore we can do a binary search of s in that sequence, and build TopLeftMin$(r, (x, y))$ for $x \in I_4$ with at most $\lceil \log_2(|I_4| + 1) \rceil \leq \log_2 |I_4| + 1 = \log_2(2|I_4|) = \log_2 |I_1|$ comparisons. For each $y \in I_2$, we do the above binary search to compute TopLeftMin$(r, (x, y))$ for $x \in I_4$. Now, all the entries in TopLeftMin(r) are filled. The total number of comparisons made is $|I_2| \log_2 |I_1|$. Therefore, the DominanceMin structures for the range r is built within $4 |I_2| \log_2 |I_1|$. In the case of $|I_1| \leq |I_2|$, the number of comparisons is $4 |I_1| \log_2 |I_2|$.

THEOREM 4.1. *The proposed 2D RMQ preprocessing algorithm uses $O(N)$ comparisons.*

Proof. The comparisons done in the whole dynamic programming algorithm can be calculated by the method below: For each $0 \leq i \leq \log m$ and $0 \leq j \leq \log n$, let $F(i, j)$ be the set of 2^i by 2^j canonical ranges. We have $|F(i, j)| = m/2^i \cdot n/2^j = N/2^{i+j}$. If $i \geq j$, then the number of comparisons made during the constructions for all 2^i by 2^j canonical ranges is $O(|F(i, j)| \cdot 2^j \log 2^i) = O(i/2^i \cdot N)$. Therefore, the number of comparisons made for constructing $\bigcup_{i \geq j} F(i, j)$ is in the order of

$$\sum_{i=0}^{\log m} \sum_{j=0}^{i} \frac{i}{2^i} N = \left(\sum_{i=0}^{\log m} \frac{i(i+1)}{2^i} \right) N$$
$$\leq \left(\sum_{i=0}^{\infty} \frac{i(i+1)}{2^i} \right) N = O(N).$$

Similarly, the number of comparisons made for constructing $\bigcup_{i \leq j} F(i, j)$ is also $O(N)$. Because CR $= \left(\bigcup_{i \leq j} F(i, j) \right) \cup \left(\bigcup_{i \geq j} F(i, j) \right)$, the total number of comparisons made for constructing the DominanceMin structures for CR is $O(N)$. ∎

4.1.2 Querying Any 2D range query q can split to at most 4 subqueries, where each subquery is a 2-side range query within some canonical range. The subqueries can be answered without any comparison by a table lookup in the DominanceMin structures of the corresponding canonical ranges, and the final query result can be computed by at most 3 comparisons.

To help find out these canonical ranges in constant time, we can do the following preprocessing in linear time and space (in RAM and without any comparison): Build auxiliary trees for CIS$_1$ and CIS$_2$ respectively as we did in the "Faster Querying" part of Section 3.1. Name the auxiliary trees T_1 and T_2 respectively. Then for any query $q = [a_1, b_1] \times [a_2, b_2]$, find the nearest common ancestor of $v_{[a_1, a_1]}$ and $v_{[b_1, b_1]}$ in T_1, and denote the resulting ancestor by v_1; find the nearest common ancestor of $v_{[a_2, a_2]}$ and $v_{[b_2, b_2]}$ in T_2, and denote the resulting ancestor by v_2. Assume that v_1 and v_2 are non-leaves (we will omit the similar discussions for the cases when either v_1 or v_2 is a leaf node). Let v_{11} and v_{12} be v_1's children in T_1, and v_{21} and v_{22} be v_2's children in T_2. Then the canonical ranges that we want to find are $I(v_{11}) \times I(v_{21})$, $I(v_{11}) \times I(v_{22})$, $I(v_{12}) \times I(v_{21})$ and $I(v_{12}) \times I(v_{22})$, where $I(v)$ represents the canonical interval that is associated with the tree node v.

THEOREM 4.2. *With the linear-comparison preprocessing for a 2D array, it takes at most 3 comparisons to answer a range minimum query.*

4.2 d-Dimensional RMQ For each $1 \leq k \leq d$, we construct the canonical interval set CIS$_k$ for $[1..n_k]$. Denote the set of canonical ranges by CR $= \prod_{k=1}^{d}$ CIS$_k$. Note that $|$CIS$_k| = 2n_k - 1$, and $|$CR$| \leq 2^d N$. Within each canonical range r, we precompute the following data structures for each index $\mathbf{x} \in r$ (as a vector) and each direction $\mathbf{u} \in \{-1, 1\}^d$ (as a vector),

$$\text{DominanceMin}(r, \mathbf{u}, \mathbf{x}) = \min_{\mathbf{i} \in r \cap \text{Dom}(\mathbf{x}, \mathbf{u})} A[\mathbf{i}],$$

where $\mathbf{i}$ is a d-length integer vector, and Dom$(\mathbf{x}, \mathbf{u})$ is defined by

$$\left\{ \mathbf{p} \;\middle|\; \begin{array}{l} \mathbf{p} \text{ is a vector that satisfies} \\ u_k(p_k - x_k) \geq 0 \text{ for } 1 \leq k \leq d \end{array} \right\}.$$

The DominanceMin structures can be computed using a dynamic programming algorithm similar to the 2D case: First sort the canonical ranges, and then do the bottom-up merging. When merging the DominanceMin structures for two neighboring canonical ranges, the dimension that has the longest length should be chosen to do the binary search in order to speed up the merging. The number of comparisons for constructing DominanceMin for a canonical range $r = I_1 \times I_2 \times \cdots \times I_d$ is $2^d |r| \frac{\log l}{l}$, where $l = \max_{1 \leq k \leq d} |I_k|$. The 2^d factor

comes from 2^d dominance directions. The $\frac{\log l}{l}$ factor improvement (over a brute force method) comes from the binary search.

As in the 2D case, a query can be decomposed into 2^d subqueries, where the subquery result can be located directly in the DominanceMin structures. The subqueries can be identified by d nearest common ancestor queries.

THEOREM 4.3. *The algorithm preprocesses a d-dimensional array in at most $B(d)N$ comparisons, where $B(d) = O\big((2.89)^d \cdot (d+1)!\big)$. The number of comparisons to answer a d-dimensional query is $2^d - 1$.*

Proof. **Preprocessing Complexity.**
Let $R(l)$ be the set of canonical ranges whose longest side has *at most* length l, where $l = 2^j$ for some j. Let $R'(l)$ be the set of canonical ranges whose longest side has *exact* length l. We have

$$(4.1) \qquad R'(l) = R(l) \setminus R(l/2).$$

Let $P'_0(l)$ be the total number of comparisons to build the DominanceMin structures for $R'(l)$ using the brute force merging algorithm (i.e., use $2^d |r|$ comparisons for each canonical range r), and $P'(l)$ be the total number of comparisons using our binary-search-enhanced merging algorithm (i.e., $2^d |r| \frac{\log l}{l}$ comparisons for each canonical range r).

Similarly, let $P_0(l)$ be the total number of comparisons to build the DominanceMin structures for $R(l)$ using the brute force merging algorithm, and $P(l)$ be the total number of comparisons using our binary-search-enhanced merging algorithm. Denote the total number of comparisons for all the canonical ranges using the enhanced merging to be $P = P(\max_{1 \le k \le d} n_k)$. We will compute below an upper bound of P.

Because there are $\prod_{1 \le k \le d} n_k / 2^{j_k}$ canonical ranges of size $2^{j_1} \times 2^{j_2} \times \cdots \times 2^{j_d}$, and each brute force merging cost $2^d \prod_{1 \le k \le d} 2^{j_k}$ comparisons, we have

$$P_0(l) = 2^d \prod_{1 \le k \le d} \sum_{0 \le j \le \log l} \frac{n_k}{2^j} \cdot 2^j$$

$$= 2^d \prod_{1 \le k \le d} \big((1 + \log l) \cdot n_k\big)$$

$$(4.2) \qquad = 2^d (1 + \log l)^d N.$$

Because of Equation (4.1), we have

$$(4.3) \qquad P'_0(l) = P_0(l) - P_0(l/2) \le P_0(l).$$

The $\frac{\log l}{l}$ factor improvement for all canonical ranges in $R'(l)$ implies

$$(4.4) \qquad P'(l) = \frac{\log l}{l} P'_0(l).$$

The total number of comparisons for the enhanced merging is

$$P = \sum_{i \ge 0} P'(2^i)$$

$$= \sum_{i \ge 0} \frac{i}{2^i} P'_0(2^i) \qquad \text{(from Equation (4.4))}$$

$$\le \sum_{i \ge 0} \frac{i}{2^i} P_0(2^i) \qquad \text{(from Inequality (4.3))}$$

$$= \sum_{i \ge 0} 2^d \frac{i}{2^i} (1+i)^d N \qquad \text{(from Equation (4.2))}.$$

Therefore,

$$B(d) = \frac{P}{N} \le 2^d \sum_{i \ge 0} \frac{i(1+i)^d}{2^i} \le 2^d \sum_{i \ge 0} \frac{(1+i)^{d+1}}{2^i}$$

$$= 2^{d+2} \sum_{i \ge 0} \frac{(1+i)^{d+1}}{2^{i+2}} \le 2^{d+2} \sum_{i \ge 0} \frac{i^{d+1}}{2^{i+1}},$$

where $\sum_{i \ge 0} \frac{i^{d+1}}{2^{i+1}}$ is finite, and it is an *ordered Bell number* [21]. The ordered Bell numbers $\tilde{b}(n)$ has the following complexity

$$\tilde{b}(n) = \sum_{i \ge 0} \frac{i^n}{2^{i+1}} = \frac{1}{2(\ln 2)^{n+1}} n! + O\big((0.16)^n n!\big)$$

$$= O\big((1.443)^n n!\big).$$

Thus, $B(d)$ is a constant that depends on d, and it has an order of $2^{d+2} \tilde{b}(d+1) = O\big((2.89)^d \cdot (d+1)!\big)$.

Querying Complexity. Any online query can be divided into 2^d subqueries, where each subquery is a dominance range min query within some canonical range. Similar to the 2D case, d nearest common ancestor queries are sufficient to find out the 2^d canonical ranges. Each subquery is answered without any comparison by checking the DominanceMin structures. Therefore, $2^d - 1$ comparisons are sufficient to answer a range min query, given the preprocessed data structures in the comparison model. ∎

5 Multidimensional RMQ in RAM Model

In this section, we will discuss the implementation of the d-dimensional RMQ in the RAM model. Throughout this section, we assume that d is a fixed constant. Also, we assume that the word size of the random access machine is $W = c \log N$ for some constant c.

Similar to the 1D case, we construct micro cubes with side length $g = (c_d \log N)^{1/d}$ where the constant

c_d is chosen to satisfy $c_d < 1/B(d)$. More specifically, each dimension is divided into intervals of length g, and the Cartesian products from the d sets of the intervals form the set of micro cubes. Note that if $g \geq n_k$ for some k, then there is no need to do any interval partition on the k_{th} dimension. We have Lemma 5.1.

LEMMA 5.1. *The total time and space (in terms of bits) complexities for constructing the micro data structures are $O(B(d)N)$. Any query that fits in a micro cube can be answered in $O(2^d)$ time.*

Proof. Let G represent the size of a micro cube, then we have $G \leq g^d = c_d \log N$. As in our 1D case, we construct a decision tree to recognize the type of a micro cube. The time and space (in bits) to construct the $B(d)G$-depth decision tree is $O(2^{B(d)G} \cdot G \log^{d+1} G)$, because there are $O(2^{B(d)G})$ tree nodes, and each node costs $O(G \log^d G)$ time and $O(G \log^{d+1} G)$ bits.

The "Indices" structures at the leaves cost a total of $O(2^{B(d)G} \cdot G^2)$ time and $O(2^{B(d)G} \cdot G^2 \log G)$ bits.

For each micro cube, we recognize its type in $O(B(d)G)$ time and store the type with $O(B(d)G)$ bits. Note that a constant number of words are enough to store the type, because $B(d)G \leq B(d)c_d \log N < \log N$. The total time and space (in bits) for computing the types of all N/G micro cubes are $O(N/G \cdot B(d)G) = O(B(d)N)$.

Summing the time and space, we have: The time complexity of constructing the micro data structures is proportional to (note that $c_d < 1/B(d)$)

$$
\begin{aligned}
B(d)N + 2^{B(d)G}G^2 &\leq B(d)N + 2^{B(d)c_d \log N}(c_d \log N)^2 \\
&= B(d)N + N^{B(d)c_d} \cdot (c_d \log N)^2 \\
&= O(B(d)N).
\end{aligned}
$$

Similarly, the space is also $O(B(d)N)$ bits.

To answer a query q that completely lies within a micro cube, locate the pre-computed type t of that micro cube, and then compare up to 2^d candidates indexed by Indices(t, q) to obtain the minimum of $A[q]$. ∎

The case when a query crosses any border of a micro cube can be solved in the following dimension reduction method (with more preprocessing). Choose a dimension where the crossing occurs; if the crossings occur in more than one dimension, then choose any of them. Without loss of generality, assume that the chosen dimension is the 1_{st} dimension. For cleaner presentation, assume that n_1 can be divided by g.

Let the intervals on the 1_{st} dimension to be $I_j = [g(j-1)+1, gj]$ for $1 \leq j \leq n/g$. Assume that the query is $q = [x_1, x_2] \times q'$, where q' is a $(d-1)$-dimensional

range. Because q spans more than one interval on the 1_{st} dimension, so $[x_1, x_2]$ must be spanning on $I_u, I_{u+1}, \cdots, I_v$, where $u = \lceil x_1/g \rceil$, $v = \lceil x_2/g \rceil$ and $u < v$. There are at most three subqueries whose union is q, which are $q_1 = (I_u \cap [x_1, x_2]) \times q'$, $q_2 = (I_v \cap [x_1, x_2]) \times q'$ and $q_3 = (\bigcup_{u < w < v} I_w) \times q'$.

The range minima of q_1 and q_2 can be solved recursively to $(d-1)$-dimensional RMQ queries. Since q_1 and q_2 are symmetric, we will only show how to do the recursion for q_1. The reduced query needs to be answered on a new $(d-1)$-dimensional array A'_{1,x_1} with size $n_2 \times n_3 \times \cdots n_d$. The entries in A'_{1,x_1} are pre-computed by the following aggregation: (set $k \leftarrow 1$ and $x \leftarrow x_1$)

$$
\begin{aligned}
&A'_{k,x}(i_1, \cdots, i_{k-1}, i_{k+1} \cdots, i_d) \\
&= \min_{x \leq i_k \leq g \lceil x/g \rceil} A(i_1, i_2, \cdots, i_d).
\end{aligned}
$$

The dimension-reduced query is q' because of the fact that $\min A[q_1] = \min A'_{1,x_1}[q']$.

As discussed above, to answer the subqueries like q_1 and q_2, we have to recursively preprocess the array A in the following way: For answering subqueries like q_1, construct $A'_{k,x}$ for each dimension $1 \leq k \leq d$ and each $x \in [1, n_k]$; similarly, for subqueries like q_2, construct $A''_{k,x}$ for each dimension $1 \leq k \leq d$ and each $x \in [1, n_k]$, where

$$
\begin{aligned}
&A''_{k,x}(i_1, \cdots, i_{k-1}, i_{k+1} \cdots, i_d) \\
&= \min_{g(\lceil x/g \rceil - 1)+1 \leq i_k \leq x} A(i_1, i_2, \cdots, i_d).
\end{aligned}
$$

Recursively preprocess the $(d-1)$-dimensional arrays $A'_{k,x}$ and $A''_{k,x}$.

The subquery q_3, if it exists, can be solved using the algorithm from Lemma 5.2. The idea is to pre-aggregate the data for each interval along the 1_{st} dimension to create a new array A_1^* of size $\frac{n_1}{g} \times n_2 \times \cdots \times n_d$, and then preprocess A_1^* using Lemma 5.2 to answer a d-dimensional range minimum query transformed from q_3. More specifically, construct the new array A_1^* according to

$$
\begin{aligned}
&A_k^*(i_1, \cdots, i_{k-1}, i_k^*, i_{k+1} \cdots, i_d) \\
&= \min_{g(i_k^*-1)+1 \leq i_k \leq g i_k^*} A(i_1, i_2, \cdots, i_d),
\end{aligned}
$$

in $O(N)$ time and preprocess it in $O(N/g \cdot (3 \log \log(N/g))^d) = o(N)$ time and space (in words). To answer q_3 on A, it is equivalent to answer $[x_1^*, x_2^*] \times q'$ on A_1^*, where $x_1^* = \lceil x_1/g \rceil$ and $x_2^* = \lceil x_2/g \rceil$. Generally speaking, to answer subqueries like q_3, we need to construct A_k^* for every $1 \leq k \leq d$, and preprocess them with the algorithm from Lemma 5.2.

LEMMA 5.2. *For any fixed d, there is an algorithm to preprocess a d-dimensional array using $O(N(3 \log \log N)^d)$ time and space (in words), and answer the range minimum query in $O(3^d)$ time.*

Proof. Consider the semigroup model, which the min operator fits in. For the 1-dimensional case, Yao [22] (see also Alon and Schieber [1]) presented an algorithm to preprocess an array using $O(n \log \log n)$ time and $O(n \log \log n)$ space (in terms of semigroup elements) to support range sum query, where the query is answered by summing 3 semigroup elements. Use their algorithm as a base scheme in the dimension reduction algorithm of Chazelle and Rosenberg [6], we then get an $O(N(3 \log \log N)^d)$ time and space preprocessing algorithm to support $O(3^d)$-time range semigroup sum query. Their algorithms can be implemented in the RAM model without changing the complexities. ∎

Putting everything together, we have Theorem 5.1.

THEOREM 5.1. *For any fixed d, there is an algorithm to preprocess a d-dimensional array using $O(B(d)N)$ time and $O(2^d d! N)$ additional space (in words), and answer the range minimum query in $O(3^d)$ time.*

Proof. Let $T(N, d)$ be the total time, $S(N, d)$ be the total additional space (in words) to preprocess the input array, and $Q(d)$ be the querying time, then we have

$$T(N, d) \leq \sum_{1 \leq k \leq d} (2n_k T(N/n_k, d - 1) + O(N))$$
$$+ O(B(d)N) \quad \text{where } \prod_{k=1}^{d} n_k = N;$$

$$T(N, 1) = O(N);$$

$$S(N, d) \leq \sum_{1 \leq k \leq d} (2n_k S(N/n_k, d - 1) + O(N))$$
$$+ O(B(d)N/W) \quad \text{where } \prod_{k=1}^{d} n_k = N;$$

$$S(N, 1) = O(N);$$
$$Q(d) \leq 2Q(d - 1) + O(3^d);$$
$$Q(1) = O(1).$$

Solve the recurrences with $B(d) = O((2.89)^d \cdot (d + 1)!)$, then we get

$$T(N, d) = O(B(d)N);$$
$$S(N, d) = O(2^d d! N + B(d)N/\log N) = O(2^d d! N);$$
$$Q(d) = O(3^d).$$

∎

References

[1] N. Alon and B. Schieber. Optimal preprocessing for answering online product queries. Technical report, Tel Aviv University, 1987.

[2] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 258–264, New York, NY, USA, 2002. ACM.

[3] A. Amir, J. Fischer, and M. Lewenstein. Two-dimensional range minimum queries. In *Proceedings of 18th Annual Symposium on Combinatorial Pattern Matching*, pages 286–294, 2007.

[4] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005.

[5] J. L. Bentley. Decomposable searching problems. *Inf. Process. Lett.*, 8(5):244–251, 1979.

[6] B. Chazelle and B. Rosenberg. Computing partial sums in multidimensional arrays. In *SCG '89: Proceedings of the fifth annual symposium on Computational geometry*, pages 131–139, New York, NY, USA, 1989. ACM.

[7] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP(On-line Analytical Processing) to User-Analysts: An IT Mandate. In *Technical Report, Codd E.F. & Associates; 1993*.

[8] E. Demaine, G. M. Landau, and O. Weimann. On Cartesian trees and range minimum queries. In *ICALP'09: Proceedings of the 36th International Colloquium on Automata, Languages and Programming*, 2009.

[9] B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Comput.*, 21(6):1184–1192, 1992.

[10] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 135–143, New York, NY, USA, 1984. ACM.

[11] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[12] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, New York, NY, USA, 1997.

[13] Y. Han. Deterministic sorting in $o(n \log \log n)$ time and linear space. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 602–608, New York, NY, USA, 2002. ACM.

[14] Y. Han and M. Thorup. Integer sorting in $0(n\sqrt{\log\log n})$ expected time and linear space. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:135, 2002.

[15] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984.

[16] J. Komlós. Linear verification for spanning trees. In *FOCS '84: Proceedings of the 25th Annual Symposium on Foundations of Computer Science, 1984.*, pages 201–206, Washington, DC, USA, 1984. IEEE Computer Society.

[17] C. K. Poon. Optimal range max datacube for fixed dimensions. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, pages 158–172, London, UK, 2002. Springer-Verlag.

[18] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.

[19] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[20] J. Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.

[21] H. S. Wilf. *Generatingfunctionology*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

[22] A. C. Yao. Space-time tradeoff for answering range queries (extended abstract). In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 128–136, New York, NY, USA, 1982. ACM.