

DATA STRUCTURES FOR RANGE MINIMUM QUERIES IN MULTIDIMENSIONAL ARRAYS ★

Hao Yuan

City University of Hong Kong

★ Joint work with Mikhail J. Atallah

Appeared in SODA 2010

December 3, 2010

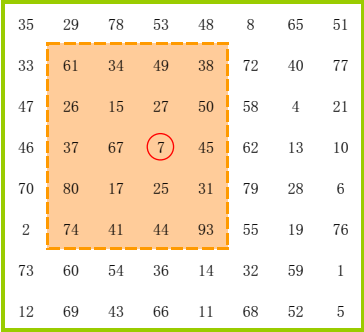
- Introduction
- Results
 - Overview
 - Details
 - Step 1: Comparison-Efficient Data Structures
 - Step 2: Random Access Machine Implementation
- Future Work
 - Online Queries
 - Offline Queries

DEFINITIONS

Given a d -dimensional array A with N entries, a *Range Minimum Query (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.,

$$\text{RMQ}(A, q) = \min A[q] = \min_{(k_1, \dots, k_d) \in q} A[k_1, \dots, k_d].$$



35	29	78	53	48	8	65	51
33	61	34	49	38	72	40	77
47	26	15	27	50	58	4	21
46	37	67	7	45	62	13	10
70	80	17	25	31	79	28	6
2	74	41	44	93	55	19	76
73	60	54	36	14	32	59	1
12	69	43	66	11	68	52	5

- **String Pattern Matching:** 1D RMQ and its related Least Common Ancestor (LCA) problems are fundamental building blocks in suffix trees/arrays
- **Computational Biology:** Finding min/max number in an alignment tableau (genome sequence analysis)
- **Image Processing:** Finding the lightest/darkest point in a range (Dilate/Erode Filter)
- **Databases:** Range Min/Max Query in OLAP Data Cube

EXAMPLE

Select the highest paid employee whose age is between 40 and 50 and joined the company during the period between 1995 and 2005

1D Range Minimum Query

- Linear Reduction to *Least Common Ancestor* (LCA) Problem
[Gabow, Bentley and Tarjan 1984]
- LCA: $O(N)$ Preprocessing, $O(1)$ Querying
[Harel and Tarjan 1984]
- RMQ & LCA: Much Studied
(Parallelization, Simplification, Distributed Algorithms, etc)
[Schieber and Vishkin 1988]
[Bender and Farach-Colton 2000]
[Alstrup et al. 2002]

PREVIOUS WORK - SEMIGROUP MODEL

Related to the semi-group sum problem (MIN is a semi-group operator)

Data Structures: $O(M)$ preprocessing time and space ($M \geq N$),
 $O(\alpha(M, N))$ querying time

- One Dimensional: [Yao 1982], [Alon and Schieber 1987]
- Multidimensional (fixed d): [Chazelle and Rosenberg 1989]

MULTIDIMENSIONAL RMQ

Unit-Cost RAM Model:

$O(1)$ cost for: Read/Write Memory, $+$, $-$, $*$, $/$, $<<$, $>>$

Comparison-Based: Array entries can only be compared

TABLE: Results for d -dimensional RMQ (d is fixed). The $O(\cdot)$ is omitted.

	Preprocess Time	Space	Querying Time
Gabow et al. 1984	$N \log^{d-1} N$	$N \log^{d-1} N$	$\log^{d-1} N$
Chazelle and Rosenberg 1989	M	M	$\alpha^d(M, N)$
Poon 2003	$N(\log^* N)^d$	N	1
Amir et al. 2007 ($d = 2$)	$N \log^{[k+1]} N$	kN	1
Our result	N	N	1

General Approach

- Design Comparison-Efficient Algorithm:
Only comparisons between input array entries are counted
- Implement the Algorithm in RAM:
All the computations are counted

Example: Minimum Spanning Tree Verification
[Komlós 1984] [Dixon, Rauch and Tarjan 1992]

Following the general approach:

- Comparison-Efficient Data Structures
 - New 1D RMQ
 - Preliminary: $O(N \log N)$ -comparison preprocessing and 1-comparison querying
 - Speedup the preprocessing to $O(N)$ comparisons
 - New data structure generalizes to two or higher dimensional cases
 - Preprocessing: $O(N)$ comparisons
 - Querying: $O(1)$ comparisons
- RAM Implementations
 - Micro blocks of size $\epsilon \log N$
 - Solve big size query by well-known algorithms
 - Solve small size query by table lookup

Following the general approach:

- Comparison-Efficient Data Structures
 - New 1D RMQ
 - Preliminary: $O(N \log N)$ -comparison preprocessing and 1-comparison querying
 - Speedup the preprocessing to $O(N)$ comparisons
 - New data structure generalizes to two or higher dimensional cases
 - Preprocessing: $O(N)$ comparisons
 - Querying: $O(1)$ comparisons
- RAM Implementations
 - Micro blocks of size $\epsilon \log N$
 - Solve big size query by well-known algorithms
 - Solve small size query by table lookup

Following the general approach:

- Comparison-Efficient Data Structures
 - New 1D RMQ
 - Preliminary: $O(N \log N)$ -comparison preprocessing and 1-comparison querying
 - Speedup the preprocessing to $O(N)$ comparisons
 - New data structure generalizes to two or higher dimensional cases
 - Preprocessing: $O(N)$ comparisons
 - Querying: $O(1)$ comparisons
- RAM Implementations
 - Micro blocks of size $\epsilon \log N$
 - Solve big size query by well-known algorithms
 - Solve small size query by table lookup

Following the general approach:

- Comparison-Efficient Data Structures
 - New 1D RMQ
 - Preliminary: $O(N \log N)$ -comparison preprocessing and 1-comparison querying
 - Speedup the preprocessing to $O(N)$ comparisons
 - New data structure generalizes to two or higher dimensional cases
 - Preprocessing: $O(N)$ comparisons
 - Querying: $O(1)$ comparisons
- RAM Implementations
 - Micro blocks of size $\epsilon \log N$
 - Solve big size query by well-known algorithms
 - Solve small size query by table lookup

Following the general approach:

- Comparison-Efficient Data Structures
 - New 1D RMQ
 - Preliminary: $O(N \log N)$ -comparison preprocessing and 1-comparison querying
 - Speedup the preprocessing to $O(N)$ comparisons
 - New data structure generalizes to two or higher dimensional cases
 - Preprocessing: $O(N)$ comparisons
 - Querying: $O(1)$ comparisons
- RAM Implementations
 - Micro blocks of size $\epsilon \log N$
 - Solve big size query by well-known algorithms
 - Solve small size query by table lookup

Following the general approach:

- Comparison-Efficient Data Structures
 - New 1D RMQ
 - Preliminary: $O(N \log N)$ -comparison preprocessing and 1-comparison querying
 - Speedup the preprocessing to $O(N)$ comparisons
 - New data structure generalizes to two or higher dimensional cases
 - Preprocessing: $O(N)$ comparisons
 - Querying: $O(1)$ comparisons
- RAM Implementations
 - Micro blocks of size $\epsilon \log N$
 - Solve big size query by well-known algorithms
 - Solve small size query by table lookup

Following the general approach:

- Comparison-Efficient Data Structures
 - New 1D RMQ
 - Preliminary: $O(N \log N)$ -comparison preprocessing and 1-comparison querying
 - Speedup the preprocessing to $O(N)$ comparisons
 - New data structure generalizes to two or higher dimensional cases
 - Preprocessing: $O(N)$ comparisons
 - Querying: $O(1)$ comparisons
- RAM Implementations
 - Micro blocks of size $\epsilon \log N$
 - Solve big size query by well-known algorithms
 - Solve small size query by table lookup

Following the general approach:

- Comparison-Efficient Data Structures
 - New 1D RMQ
 - Preliminary: $O(N \log N)$ -comparison preprocessing and 1-comparison querying
 - Speedup the preprocessing to $O(N)$ comparisons
 - New data structure generalizes to two or higher dimensional cases
 - Preprocessing: $O(N)$ comparisons
 - Querying: $O(1)$ comparisons
- RAM Implementations
 - Micro blocks of size $\epsilon \log N$
 - Solve big size query by well-known algorithms
 - Solve small size query by table lookup

Following the general approach:

- Comparison-Efficient Data Structures
 - New 1D RMQ
 - Preliminary: $O(N \log N)$ -comparison preprocessing and 1-comparison querying
 - Speedup the preprocessing to $O(N)$ comparisons
 - New data structure generalizes to two or higher dimensional cases
 - Preprocessing: $O(N)$ comparisons
 - Querying: $O(1)$ comparisons
- RAM Implementations
 - Micro blocks of size $\epsilon \log N$
 - Solve big size query by well-known algorithms
 - Solve small size query by table lookup

Following the general approach:

- Comparison-Efficient Data Structures
 - New 1D RMQ
 - Preliminary: $O(N \log N)$ -comparison preprocessing and 1-comparison querying
 - Speedup the preprocessing to $O(N)$ comparisons
 - New data structure generalizes to two or higher dimensional cases
 - Preprocessing: $O(N)$ comparisons
 - Querying: $O(1)$ comparisons
- RAM Implementations
 - Micro blocks of size $\epsilon \log N$
 - Solve big size query by well-known algorithms
 - Solve small size query by table lookup

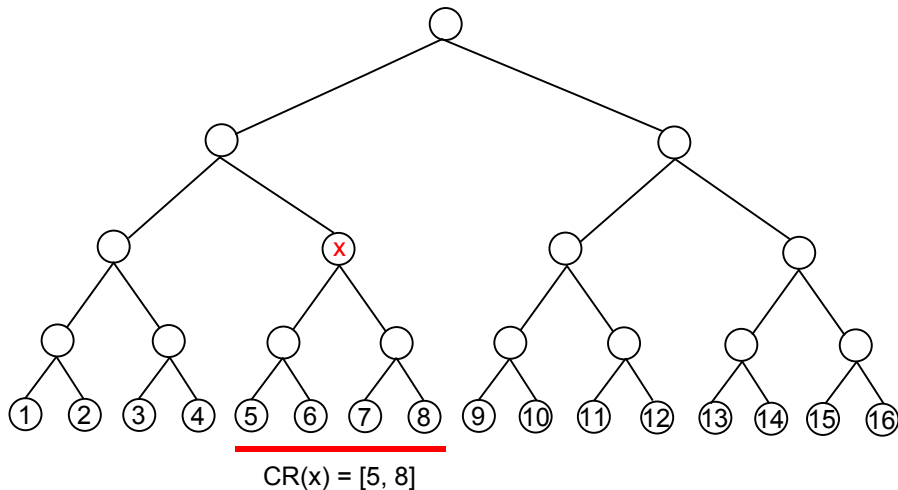
Following the general approach:

- Comparison-Efficient Data Structures
 - New 1D RMQ
 - Preliminary: $O(N \log N)$ -comparison preprocessing and 1-comparison querying
 - Speedup the preprocessing to $O(N)$ comparisons
 - New data structure generalizes to two or higher dimensional cases
 - Preprocessing: $O(N)$ comparisons
 - Querying: $O(1)$ comparisons
- RAM Implementations
 - Micro blocks of size $\epsilon \log N$
 - Solve big size query by well-known algorithms
 - Solve small size query by table lookup

If only count comparisons:

- 2D RMQ Lower Bound: [Demaine, Landau and Weimann, 2009]
If *NO COMPARISON* is allowed at the query stage, then $\Omega(N \log N)$ comparisons preprocessing is required
- **Our Result:** $O(2.89^d(d+1)!N)$ comparisons preprocessing,
 $2^d - 1$ comparisons querying

CANONICAL RANGES

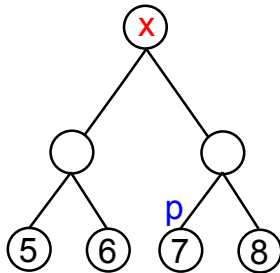


PRE-COMPUTATIONS

For each $p \in \text{CR}(x)$, define

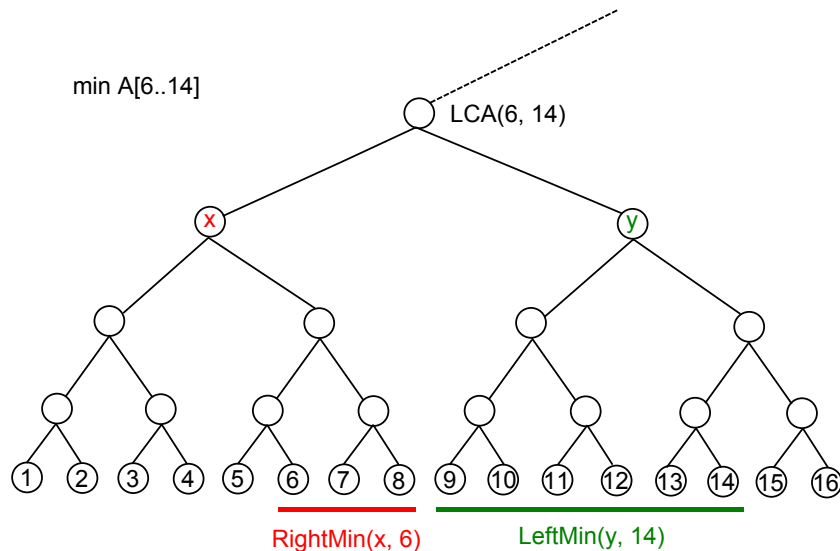
$$\text{LeftMin}(x, p) = \min_{k \in \text{CR}(x) \text{ and } k \leq p} A[k]$$

$$\text{RightMin}(x, p) = \min_{k \in \text{CR}(x) \text{ and } k \geq p} A[k]$$



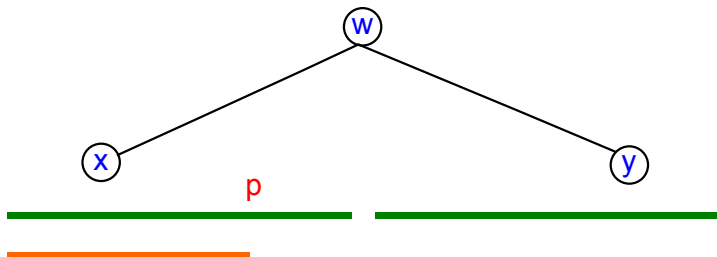
$$\text{LeftMin}(x, 7) = \min\{A[5], A[6], A[7]\}$$

$$\text{RightMin}(x, 7) = \min\{A[7], A[8]\}$$



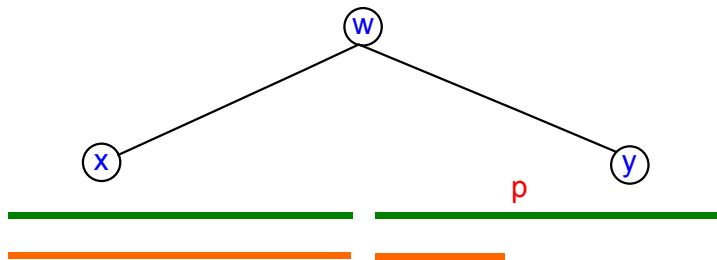
- Naïve Algorithm: $O(N \log N)$ Comparisons
- Faster Approach
 - Sort the canonical ranges by their lengths
 - Compute the LeftMin and RightMin entries for canonical ranges in the sorted order
 - For length-one canonical range $CR(w)$,
 $LeftMin(w, p) = RightMin(w, p) = A[p]$ ($p \in CR(w)$)
 - For a canonical range $CR(w)$ covering more than one position, compute the LeftMin and RightMin arrays in $O(\log |CR(w)|)$ time (instead of $O(|CR(w)|)$)

Case 1: $p \in \text{CR}(x)$



$$\text{LeftMin}(w, p) = \text{LeftMin}(x, p)$$

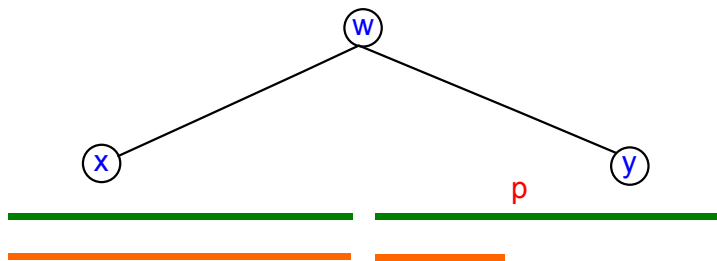
Case 2: $p \in \text{CR}(y)$



$$\text{LeftMin}(w, p) = \min \{ \min \text{CR}(x), \text{LeftMin}(y, p) \}$$

PRE-COMPUTATIONS

Case 2: $p \in \text{CR}(y)$

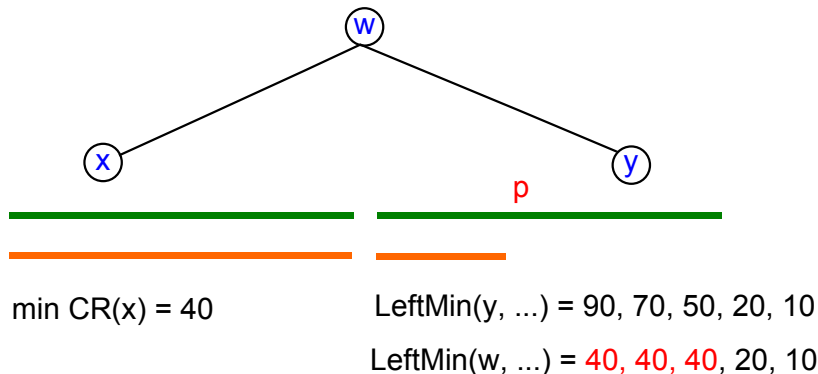


$$\text{LeftMin}(w, p) = \min \{ \min \text{CR}(x), \text{LeftMin}(y, p) \}$$

Monotonicity (Non-Increasing): $\text{LeftMin}(y, p) \geq \text{LeftMin}(y, p + 1)$

Binary Search!

Example



COMPARISON COMPLEXITY

$T(n)$: the number of comparisons to compute the LeftMin and RightMin entries for canonical ranges whose size is at most n

$$T(1) = 0$$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(\log n) \quad \text{for } n \geq 2$$

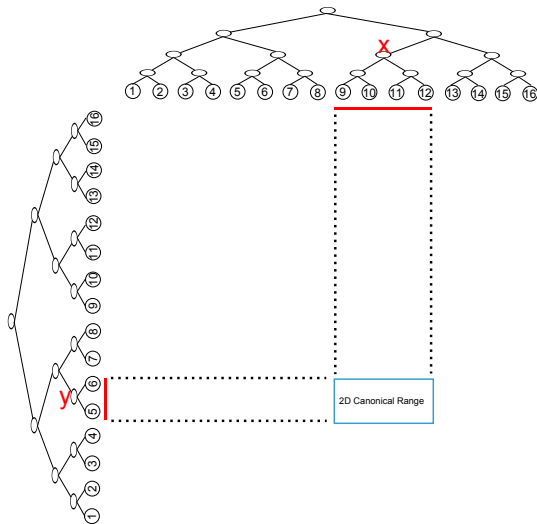
We have the preprocessing comparison complexity

$$T(n) = O(n),$$

and need to do 1 comparison at the query stage.

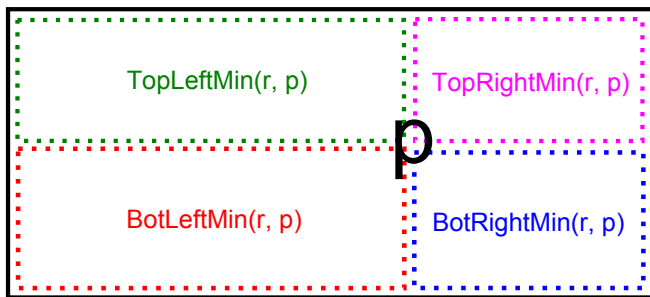
2D CASE

2D Canonical Range: Cartesian Product of Two 1D Canonical Ranges



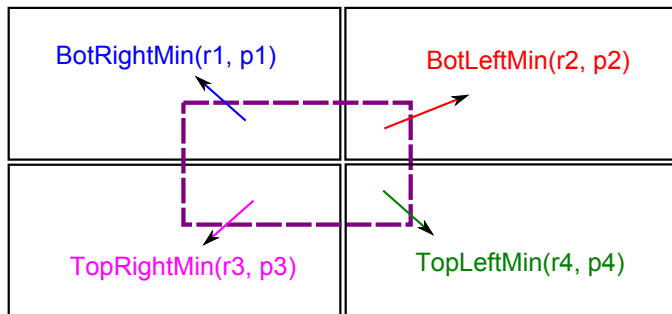
2D PRE-COMPUTATIONS

For each 2D canonical range r and a point $p \in r$, compute the 4 “*Dominance Min*” array entries



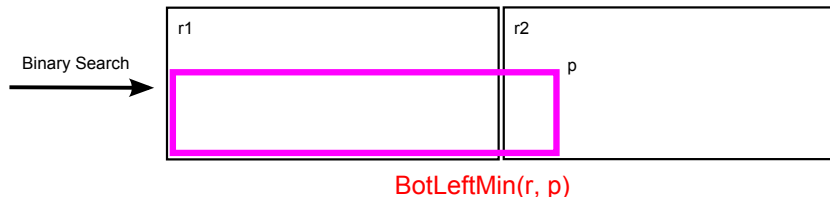
2D QUERY

For any query range q , we can always divide it into 4 parts, which are all pre-computed



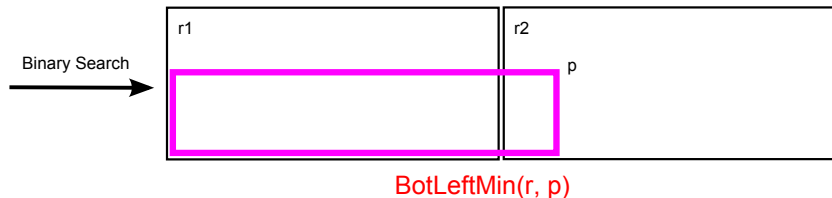
EFFICIENT PRE-COMPUTATIONS

- For any canonical range r , cut the middle of its **longer side** to obtain two smaller canonical ranges r_1 and r_2
- Do binary search row by row (or column by column)
- $O(N)$ comparisons for 2D preprocessing
- Generalize to any fixed dimension d :
 - Preprocess: $O(2.89^d(d+1)!N)$ comparisons
 - Query: $2^d - 1$ comparisons



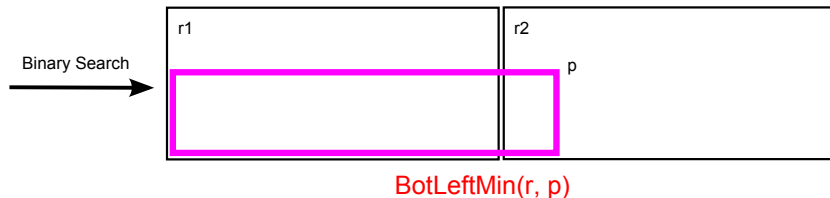
EFFICIENT PRE-COMPUTATIONS

- For any canonical range r , cut the middle of its **longer side** to obtain two smaller canonical ranges r_1 and r_2
- Do binary search row by row (or column by column)
- $O(N)$ comparisons for 2D preprocessing
- Generalize to any fixed dimension d :
 - Preprocess: $O(2.89^d(d+1)!N)$ comparisons
 - Query: $2^d - 1$ comparisons



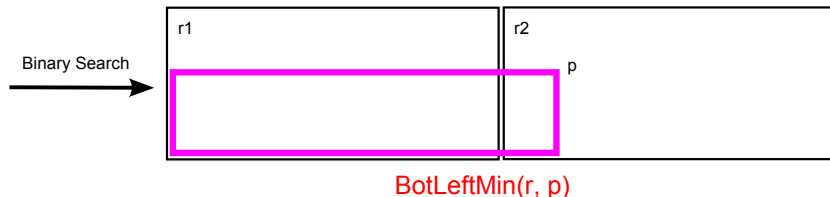
EFFICIENT PRE-COMPUTATIONS

- For any canonical range r , cut the middle of its **longer side** to obtain two smaller canonical ranges r_1 and r_2
- Do binary search row by row (or column by column)
- $O(N)$ comparisons for 2D preprocessing
- Generalize to any fixed dimension d :
 - Preprocess: $O(2.89^d(d+1)!N)$ comparisons
 - Query: $2^d - 1$ comparisons



EFFICIENT PRE-COMPUTATIONS

- For any canonical range r , cut the middle of its **longer side** to obtain two smaller canonical ranges r_1 and r_2
- Do binary search row by row (or column by column)
- $O(N)$ comparisons for 2D preprocessing
- Generalize to any fixed dimension d :
 - Preprocess: $O(2.89^d(d+1)!N)$ comparisons
 - Query: $2^d - 1$ comparisons

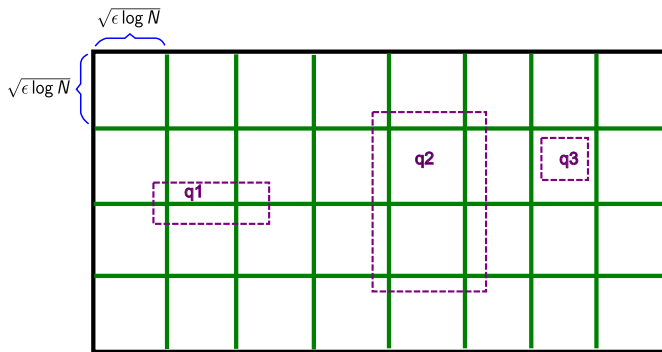


OVERVIEW OF RAM IMPLEMENTATIONS

Divide the array into micro blocks of size $\epsilon \log N$

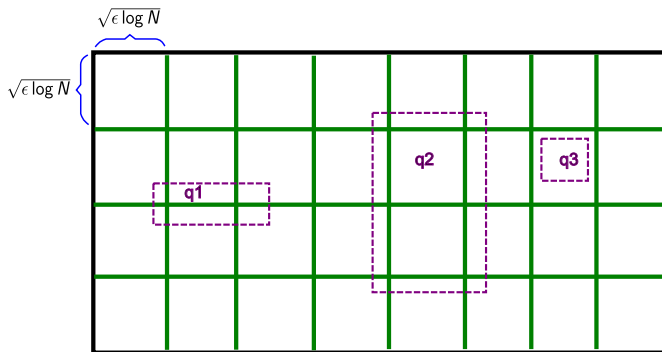
Each block is a d -dimensional cube, with side length $(\epsilon \log N)^{\frac{1}{d}}$

For example in 2D, make each block $\sqrt{\epsilon \log N}$ by $\sqrt{\epsilon \log N}$



OVERVIEW OF RAM IMPLEMENTATIONS

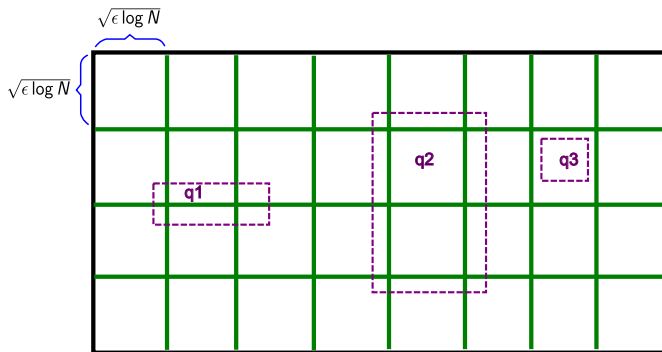
For query that crosses the border of any micro block: there exists $O(N)$ -time preprocessing and constant-time querying data structures to solve it, using dimension reductions and the help of the data structures in [Yao 1982] [Chazelle and Rosenberg 1989]



OVERVIEW OF RAM IMPLEMENTATIONS

For query that is complete within a micro block, use table lookup technique (Four Russian's Trick) to get the locations of at most 2^d candidates to compare at the querying stage

Based on our linear-comparison preprocessing data structure



Key Idea: If two micro blocks have the **same type**, then they should share the **same data structures**

- Type of a micro block: Comparison results (true/false sequence) of the linear-comparison preprocessing algorithm
- Assume $c\epsilon \log N$ comparisons to preprocess a block: at most $2^{c\epsilon \log N} = N^{c\epsilon}$ possible types
 - Choose $\epsilon < \frac{1}{c}$, then there are only a **sublinear** number of types:

$$N^{c\epsilon} \text{polylog}(\epsilon \log N) = o(N)$$

- Recognizing the types for all micro blocks in linear time:
Build a **linear-depth decision tree** according to the linear-comparison preprocessing algorithm

SUMMARY OF RAM

Our unit-cost RAM data structure

- Preprocess in $O(2.89^d(d+1)!N)$ time and $(2^d d!N)$ space
- Query in $O(3^d)$ time

FUTURE WORK - ONLINE QUERIES

- Reduce the large hidden constant (e.g., $O(2.89^d(d+1)!)$) for high dimensional cases
- Time-Space Tradeoffs
[Brodal, Davoodi and Rao, 2010]
 - Space: $O(N/c)$ bits in addition to the input
 - Time: $O(c \log^2 c)$
- Dynamic Updates
 - Remove the inverse-Ackermann factor of previous work [Poon, 2003]
 - Consider a restricted type of update - decrease only
- Extend our results to the external memory model
 - Optimal cache-oblivious data structure in 1D
[Demaine, Landau and Weimann, 2009]
 - The 2D case is still open
- Parallel preprocessing

FUTURE WORK - ONLINE QUERIES

- Reduce the large hidden constant (e.g., $O(2.89^d(d+1)!)$) for high dimensional cases
- Time-Space Tradeoffs
[Brodal, Davoodi and Rao, 2010]
 - Space: $O(N/c)$ bits in addition to the input
 - Time: $O(c \log^2 c)$
- Dynamic Updates
 - Remove the inverse-Ackermann factor of previous work [Poon, 2003]
 - Consider a restricted type of update - decrease only
- Extend our results to the external memory model
 - Optimal cache-oblivious data structure in 1D [Demaine, Landau and Weimann, 2009]
 - The 2D case is still open
- Parallel preprocessing

FUTURE WORK - ONLINE QUERIES

- Reduce the large hidden constant (e.g., $O(2.89^d(d+1)!)$) for high dimensional cases
- Time-Space Tradeoffs
[Brodal, Davoodi and Rao, 2010]
 - Space: $O(N/c)$ bits in addition to the input
 - Time: $O(c \log^2 c)$
- Dynamic Updates
 - Remove the inverse-Ackermann factor of previous work [Poon, 2003]
 - Consider a restricted type of update - decrease only
- Extend our results to the external memory model
 - Optimal cache-oblivious data structure in 1D [Demaine, Landau and Weimann, 2009]
 - The 2D case is still open
- Parallel preprocessing

FUTURE WORK - ONLINE QUERIES

- Reduce the large hidden constant (e.g., $O(2.89^d(d+1)!)$) for high dimensional cases
- Time-Space Tradeoffs
[Brodal, Davoodi and Rao, 2010]
 - Space: $O(N/c)$ bits in addition to the input
 - Time: $O(c \log^2 c)$
- Dynamic Updates
 - Remove the inverse-Ackermann factor of previous work [Poon, 2003]
 - Consider a restricted type of update - decrease only
- Extend our results to the external memory model
 - Optimal cache-oblivious data structure in 1D [Demaine, Landau and Weimann, 2009]
 - The 2D case is still open
- Parallel preprocessing

FUTURE WORK - ONLINE QUERIES

- Reduce the large hidden constant (e.g., $O(2.89^d(d+1)!)$) for high dimensional cases
- Time-Space Tradeoffs
[Brodal, Davoodi and Rao, 2010]
 - Space: $O(N/c)$ bits in addition to the input
 - Time: $O(c \log^2 c)$
- Dynamic Updates
 - Remove the inverse-Ackermann factor of previous work [Poon, 2003]
 - Consider a restricted type of update - decrease only
- Extend our results to the external memory model
 - Optimal cache-oblivious data structure in 1D [Demaine, Landau and Weimann, 2009]
 - The 2D case is still open
- Parallel preprocessing

FUTURE WORK - ONLINE QUERIES

- Reduce the large hidden constant (e.g., $O(2.89^d(d+1)!)$) for high dimensional cases
- Time-Space Tradeoffs
[Brodal, Davoodi and Rao, 2010]
 - Space: $O(N/c)$ bits in addition to the input
 - Time: $O(c \log^2 c)$
- Dynamic Updates
 - Remove the inverse-Ackermann factor of previous work [Poon, 2003]
 - Consider a restricted type of update - decrease only
- Extend our results to the external memory model
 - Optimal cache-oblivious data structure in 1D
[Demaine, Landau and Weimann, 2009]
 - The 2D case is still open
- Parallel preprocessing

FUTURE WORK - ONLINE QUERIES

- Reduce the large hidden constant (e.g., $O(2.89^d(d+1)!)$) for high dimensional cases
- Time-Space Tradeoffs
[Brodal, Davoodi and Rao, 2010]
 - Space: $O(N/c)$ bits in addition to the input
 - Time: $O(c \log^2 c)$
- Dynamic Updates
 - Remove the inverse-Ackermann factor of previous work [Poon, 2003]
 - Consider a restricted type of update - decrease only
- Extend our results to the external memory model
 - Optimal cache-oblivious data structure in 1D
[Demaine, Landau and Weimann, 2009]
 - The 2D case is still open
- Parallel preprocessing

FUTURE WORK - ONLINE QUERIES

- Reduce the large hidden constant (e.g., $O(2.89^d(d+1)!)$) for high dimensional cases
- Time-Space Tradeoffs
[Brodal, Davoodi and Rao, 2010]
 - Space: $O(N/c)$ bits in addition to the input
 - Time: $O(c \log^2 c)$
- Dynamic Updates
 - Remove the inverse-Ackermann factor of previous work [Poon, 2003]
 - Consider a restricted type of update - decrease only
- Extend our results to the external memory model
 - Optimal cache-oblivious data structure in 1D
[Demaine, Landau and Weimann, 2009]
 - The 2D case is still open
- Parallel preprocessing

FUTURE WORK - ONLINE QUERIES

- Reduce the large hidden constant (e.g., $O(2.89^d(d+1)!)$) for high dimensional cases
- Time-Space Tradeoffs
[Brodal, Davoodi and Rao, 2010]
 - Space: $O(N/c)$ bits in addition to the input
 - Time: $O(c \log^2 c)$
- Dynamic Updates
 - Remove the inverse-Ackermann factor of previous work [Poon, 2003]
 - Consider a restricted type of update - decrease only
- Extend our results to the external memory model
 - Optimal cache-oblivious data structure in 1D
[Demaine, Landau and Weimann, 2009]
 - The 2D case is still open
- Parallel preprocessing

FUTURE WORK - ONLINE QUERIES

- Reduce the large hidden constant (e.g., $O(2.89^d(d+1)!)$) for high dimensional cases
- Time-Space Tradeoffs
[Brodal, Davoodi and Rao, 2010]
 - Space: $O(N/c)$ bits in addition to the input
 - Time: $O(c \log^2 c)$
- Dynamic Updates
 - Remove the inverse-Ackermann factor of previous work [Poon, 2003]
 - Consider a restricted type of update - decrease only
- Extend our results to the external memory model
 - Optimal cache-oblivious data structure in 1D
[Demaine, Landau and Weimann, 2009]
 - The 2D case is still open
- Parallel preprocessing

FUTURE WORK - ONLINE QUERIES

- Reduce the large hidden constant (e.g., $O(2.89^d(d+1)!)$) for high dimensional cases
- Time-Space Tradeoffs
[Brodal, Davoodi and Rao, 2010]
 - Space: $O(N/c)$ bits in addition to the input
 - Time: $O(c \log^2 c)$
- Dynamic Updates
 - Remove the inverse-Ackermann factor of previous work [Poon, 2003]
 - Consider a restricted type of update - decrease only
- Extend our results to the external memory model
 - Optimal cache-oblivious data structure in 1D
[Demaine, Landau and Weimann, 2009]
 - The 2D case is still open
- Parallel preprocessing

FUTURE WORK - OFFLINE QUERIES

Compute the minimums within a sliding window:

- Given the window size p and an input sequence x_1, x_2, \dots, x_n , compute for each $1 \leq i \leq n$

$$y_i = \min_{0 \leq k < p} x_{i+k}$$



- Generalization to multidimensional cases: Compute minimums in a sliding rectangle or box area
- It is a basic filter in signal processing and mathematical morphology
Keyword: Erosion and Dilation Filters

FUTURE WORK - OFFLINE QUERIES

Compute the minimums within a sliding window:

- Given the window size p and an input sequence x_1, x_2, \dots, x_n , compute for each $1 \leq i \leq n$

$$y_i = \min_{0 \leq k < p} x_{i+k}$$



- Generalization to multidimensional cases: Compute minimums in a sliding rectangle or box area
- It is a basic filter in signal processing and mathematical morphology
Keyword: Erosion and Dilation Filters

FUTURE WORK - OFFLINE QUERIES

Compute the minimums within a sliding window:

- Given the window size p and an input sequence x_1, x_2, \dots, x_n , compute for each $1 \leq i \leq n$

$$y_i = \min_{0 \leq k < p} x_{i+k}$$



- Generalization to multidimensional cases: Compute minimums in a sliding rectangle or box area
- It is a basic filter in signal processing and mathematical morphology
Keyword: Erosion and Dilation Filters

FUTURE WORK - OFFLINE QUERIES

The comparison complexity of the running minimum filters

- Measured in the average number of comparisons per pixel (when n is very large)
- 1D Case
 - [Gil and Kimmel, 2002]: $1.5 + O(\frac{\log p}{p})$
 - [Yuan and Atallah]: $1 + O(\frac{1}{\sqrt{p}})$
 - Trivial lower bound: 1 comparison per pixel
- Multidimensional Cases
 - Assume C_1 is the best upper bound in the 1D case
 - d -dimensional case: dC_1 comparisons per pixel [Gil and Werman, 1993]
 - Can we do 2 or less comparisons per pixel for 2D case?
 - Non-trivial lower bound?
 - Cache-efficient or cache-oblivious implementation?

FUTURE WORK - OFFLINE QUERIES

The comparison complexity of the running minimum filters

- Measured in the average number of comparisons per pixel (when n is very large)
- 1D Case
 - [Gil and Kimmel, 2002]: $1.5 + O(\frac{\log p}{p})$
 - [Yuan and Atallah]: $1 + O(\frac{1}{\sqrt{p}})$
 - Trivial lower bound: 1 comparison per pixel
- Multidimensional Cases
 - Assume C_1 is the best upper bound in the 1D case
 - d -dimensional case: dC_1 comparisons per pixel [Gil and Werman, 1993]
 - Can we do 2 or less comparisons per pixel for 2D case?
 - Non-trivial lower bound?
 - Cache-efficient or cache-oblivious implementation?

FUTURE WORK - OFFLINE QUERIES

The comparison complexity of the running minimum filters

- Measured in the average number of comparisons per pixel (when n is very large)
- 1D Case
 - [Gil and Kimmel, 2002]: $1.5 + O(\frac{\log p}{p})$
 - [Yuan and Atallah]: $1 + O(\frac{1}{\sqrt{p}})$
 - Trivial lower bound: 1 comparison per pixel
- Multidimensional Cases
 - Assume C_1 is the best upper bound in the 1D case
 - d -dimensional case: dC_1 comparisons per pixel [Gil and Werman, 1993]
 - Can we do 2 or less comparisons per pixel for 2D case?
 - Non-trivial lower bound?
 - Cache-efficient or cache-oblivious implementation?

FUTURE WORK - OFFLINE QUERIES

The comparison complexity of the running minimum filters

- Measured in the average number of comparisons per pixel (when n is very large)
- 1D Case
 - [Gil and Kimmel, 2002]: $1.5 + O(\frac{\log p}{p})$
 - [Yuan and Atallah]: $1 + O(\frac{1}{\sqrt{p}})$
 - Trivial lower bound: 1 comparison per pixel
- Multidimensional Cases
 - Assume C_1 is the best upper bound in the 1D case
 - d -dimensional case: dC_1 comparisons per pixel [Gil and Werman, 1993]
 - Can we do 2 or less comparisons per pixel for 2D case?
 - Non-trivial lower bound?
 - Cache-efficient or cache-oblivious implementation?

FUTURE WORK - OFFLINE QUERIES

The comparison complexity of the running minimum filters

- Measured in the average number of comparisons per pixel (when n is very large)
- 1D Case
 - [Gil and Kimmel, 2002]: $1.5 + O(\frac{\log p}{p})$
 - [Yuan and Atallah]: $1 + O(\frac{1}{\sqrt{p}})$
 - Trivial lower bound: 1 comparison per pixel
- Multidimensional Cases
 - Assume C_1 is the best upper bound in the 1D case
 - d -dimensional case: dC_1 comparisons per pixel [Gil and Werman, 1993]
 - Can we do 2 or less comparisons per pixel for 2D case?
 - Non-trivial lower bound?
 - Cache-efficient or cache-oblivious implementation?

FUTURE WORK - OFFLINE QUERIES

The comparison complexity of the running minimum filters

- Measured in the average number of comparisons per pixel (when n is very large)
- 1D Case
 - [Gil and Kimmel, 2002]: $1.5 + O(\frac{\log p}{p})$
 - [Yuan and Atallah]: $1 + O(\frac{1}{\sqrt{p}})$
 - Trivial lower bound: 1 comparison per pixel
- Multidimensional Cases
 - Assume C_1 is the best upper bound in the 1D case
 - d -dimensional case: dC_1 comparisons per pixel [Gil and Werman, 1993]
 - Can we do 2 or less comparisons per pixel for 2D case?
 - Non-trivial lower bound?
 - Cache-efficient or cache-oblivious implementation?

FUTURE WORK - OFFLINE QUERIES

The comparison complexity of the running minimum filters

- Measured in the average number of comparisons per pixel (when n is very large)
- 1D Case
 - [Gil and Kimmel, 2002]: $1.5 + O(\frac{\log p}{p})$
 - [Yuan and Atallah]: $1 + O(\frac{1}{\sqrt{p}})$
 - Trivial lower bound: 1 comparison per pixel
- Multidimensional Cases
 - Assume C_1 is the best upper bound in the 1D case
 - d -dimensional case: dC_1 comparisons per pixel [Gil and Werman, 1993]
 - Can we do 2 or less comparisons per pixel for 2D case?
 - Non-trivial lower bound?
 - Cache-efficient or cache-oblivious implementation?

FUTURE WORK - OFFLINE QUERIES

The comparison complexity of the running minimum filters

- Measured in the average number of comparisons per pixel (when n is very large)
- 1D Case
 - [Gil and Kimmel, 2002]: $1.5 + O(\frac{\log p}{p})$
 - [Yuan and Atallah]: $1 + O(\frac{1}{\sqrt{p}})$
 - Trivial lower bound: 1 comparison per pixel
- Multidimensional Cases
 - Assume C_1 is the best upper bound in the 1D case
 - d -dimensional case: dC_1 comparisons per pixel [Gil and Werman, 1993]
 - Can we do 2 or less comparisons per pixel for 2D case?
 - Non-trivial lower bound?
 - Cache-efficient or cache-oblivious implementation?

FUTURE WORK - OFFLINE QUERIES

The comparison complexity of the running minimum filters

- Measured in the average number of comparisons per pixel (when n is very large)
- 1D Case
 - [Gil and Kimmel, 2002]: $1.5 + O(\frac{\log p}{p})$
 - [Yuan and Atallah]: $1 + O(\frac{1}{\sqrt{p}})$
 - Trivial lower bound: 1 comparison per pixel
- Multidimensional Cases
 - Assume C_1 is the best upper bound in the 1D case
 - d -dimensional case: dC_1 comparisons per pixel [Gil and Werman, 1993]
 - Can we do 2 or less comparisons per pixel for 2D case?
 - Non-trivial lower bound?
 - Cache-efficient or cache-oblivious implementation?

FUTURE WORK - OFFLINE QUERIES

The comparison complexity of the running minimum filters

- Measured in the average number of comparisons per pixel (when n is very large)
- 1D Case
 - [Gil and Kimmel, 2002]: $1.5 + O(\frac{\log p}{p})$
 - [Yuan and Atallah]: $1 + O(\frac{1}{\sqrt{p}})$
 - Trivial lower bound: 1 comparison per pixel
- Multidimensional Cases
 - Assume C_1 is the best upper bound in the 1D case
 - d -dimensional case: dC_1 comparisons per pixel [Gil and Werman, 1993]
 - Can we do 2 or less comparisons per pixel for 2D case?
 - Non-trivial lower bound?
 - Cache-efficient or cache-oblivious implementation?

FUTURE WORK - OFFLINE QUERIES

The comparison complexity of the running minimum filters

- Measured in the average number of comparisons per pixel (when n is very large)
- 1D Case
 - [Gil and Kimmel, 2002]: $1.5 + O(\frac{\log p}{p})$
 - [Yuan and Atallah]: $1 + O(\frac{1}{\sqrt{p}})$
 - Trivial lower bound: 1 comparison per pixel
- Multidimensional Cases
 - Assume C_1 is the best upper bound in the 1D case
 - d -dimensional case: dC_1 comparisons per pixel [Gil and Werman, 1993]
 - Can we do 2 or less comparisons per pixel for 2D case?
 - Non-trivial lower bound?
 - Cache-efficient or cache-oblivious implementation?

END

Thank You!