Range Minimum Query (RMQ)

Input: \( A = \{ A[1], A[2], \ldots, A[n] \} \) — array of size \( n \).

Task: precompute \( A \), so that we can answer queries

\[
\text{range-min}(i,j) \rightarrow \min \{ A[k] \mid i \leq k \leq j \}
\]

(probably return index \( k \) — \( k \) s.t. \( A[k] < A[k'] \) for all \( i \leq k' < j \))

E.g.: \[123168147\]

\[
\begin{align*}
\text{range-min}(2,4) & \rightarrow & 2 & A[2]=3 \\
\text{range-min}(3,5) & \rightarrow & 3 & \text{MISS}=6 \\
\end{align*}
\]

- Static case (array fixed, no insert/delete)
  - Some solutions can be made dynamic
- Assume array entries are distinct (otherwise by random \( 0,1 \))

Quality of an algorithm measured as:

- preprocessing time \( p(n) \)
- query time \( q(n) \)

\[ \text{expected} = \langle p(n), q(n) \rangle \]

Best we can hope for: \( \langle O(n), O(1) \rangle \)

\[
\begin{align*}
\text{must at least} & \quad \text{best possible} \\
\text{read the array} & \quad \text{read the array}
\end{align*}
\]

(We will get there.)

Observation:

If we want range-sum, then easy.

\[
\text{range-sum}(i,j) \rightarrow \text{return } A[i] + \ldots + A[j]
\]

Idea: precompute all prefix-sums \( S \)

\[
\begin{align*}
S[0] &= 0 \\
S[i] &= A[i] \\
\end{align*}
\]

\[
\begin{align*}
S[n] &= S[n-1] + A[n] & \text{Overall } O(n) \text{ time}
\end{align*}
\]
\[
\text{range-sum} (i, j) \quad \text{return } (S[j] - S[i-1])
\]

\[
\text{inverse of } \langle O(n), O(n) \rangle
\]

\[
\text{range-min more difficult} \quad \langle O(n), O(n) \rangle
\]

Why solve RMQ (Applications):
- stay algorithms (pattern matching)
- algorithms on trees/graphs
- succinct data structures

**Approach 1. Precompute everything**

Compute \(M[i, j] = \text{answer to range-min}(i, j)\)

Naively: \(O(n^3)\)

Dynamic Programming: \(O(n^2)\)

\[
M[i, j] = A[i] \\
\pi[i, j] = \min \left\{ M[i, j-1], A[i], M[i+1, j] \right\}
\]

**Approach 2. Precompute nothing**

\[
\text{range-min}(i, j) \quad \text{for } k = i+1, \ldots, j \\
\quad m = \min \{ m, A[k] \} \\
\text{return } m
\]

**Running Time:** \(\langle O(n), O(n) \rangle\)
Approach 3: Augmented trees

Build a balanced BST, using the indices as keys.

- Assume \( n = 2^k \)
- Augment nodes with
  - such aging entry away
  - that will never in subtree
  (also store index of min)

- Tree size \( 2n-1 \)
- Augmentation can be computed in \( O(n) \) time
  - answer query range-min \( (i, j) \)
    - search for \( j \)
    - \( \text{lca}(i, j) \)
      - take min over \( \text{range-min} \)
      - for all nodes on path
      - with trees hugging inside
      - \( O(1) \) itself
      - \( O(1) \) overall
        - \( (\text{depth} \leq \log_2 n + 1) \)

Overall running time \( \langle O(n), O(\log n) \rangle \)

(For now, not yet optimal, we want \( \langle O(n), O(n) \rangle \))

Detour: an application of RMQ

Find lowest common ancestor (LCA) in a tree in \( O(1) \) time

\( \text{LCA}(a, i) \rightarrow \text{root} \)
\( \text{LCA}(b, i) \rightarrow a \)
\( \text{LCA}(g, i) \rightarrow a \)
\( \text{LCA}(t, i) \rightarrow \)
\[ \text{dist}(x, y) = \text{depth}(x) + \text{depth}(y) - 2 \times \text{depth}(\text{LCA}(x, y)) \]

Solve LCA problem using RMQ

- Do a "DFS" on the tree, record it on an array.
- RMQ on the array

- For every traversed edge, record both end points.
  (Euler tour)

- \text{ET} \quad \text{abacghjqicadefdca}

- \text{depth} \quad \text{01232321210121210}

\[
\text{LCA}(x, y) \rightarrow \text{return range-min}(x, y)
\]

- e.g. \text{LCA}(h, i) \rightarrow c

\[
\text{LCA}(g, c) \rightarrow a
\]

- Minimum according to depth \text{D}
- Index of some occurrence of \text{x,y} in \text{ET}

\[
O(n) \times O(n) \rightarrow \text{RMQ} \quad \text{for} \quad O(n) \times O(n) \rightarrow \text{LCA}
\]

Solve RMQ using LCA

- Recall: treap or Cartesian tree
- Tree with nodes \((x_i, y_i)\)
- \text{treap} with order \((x_i, y_i)\), \text{BST} according to \(x_i\) values
  and \text{treap} according to \(y_i\) values

- \text{treap}
  \(1 \rightarrow 5 \rightarrow 10 \rightarrow 15 \rightarrow 20 \rightarrow 25 \rightarrow 30 \rightarrow 35 \rightarrow 40 \rightarrow 45 \rightarrow 50 \rightarrow 55 \rightarrow 60 \rightarrow 65 \rightarrow 70 \rightarrow 75 \rightarrow 80 \rightarrow 85 \rightarrow 90 \rightarrow 95 \rightarrow 100 \)

- \text{treap}
  \(1 \rightarrow 5 \rightarrow 10 \rightarrow 15 \rightarrow 20 \rightarrow 25 \rightarrow 30 \rightarrow 35 \rightarrow 40 \rightarrow 45 \rightarrow 50 \rightarrow 55 \rightarrow 60 \rightarrow 65 \rightarrow 70 \rightarrow 75 \rightarrow 80 \rightarrow 85 \rightarrow 90 \rightarrow 95 \rightarrow 100 \)
given array \( A[1], \ldots, A[n] \)

Build a heap on pairs \((i, A[i])\)

```
\[\begin{array}{c}
\text{A[5]} \\
\text{A[2]} \\
\text{A[3]} \\
\text{A[4]} \\
\text{A[6]} \\
\text{A[7]} \\
\text{A[8]} \\
\text{A[9]} \\
\end{array}\]
```

```
(2) -> (5) -> (3) -> (4) -> (6) -> (7) -> (8) -> (9)
```

It follows that

\( O(\log n) \) for \( \text{LCA} \) \( \Rightarrow \) \( O(\log n) \) for \( \text{RMA} \)

* build heap

* answer LCA queries on it

**Proof (claim)**

(i) \( \text{LCA}(i, j) \) is between \( i \) and \( j \).

(ii) Subtree of \( \text{LCA}(i, j) \) contains entire interval \([i, j]\).

(iii) \( \text{LCA}(i, j) \) has minimum value in all its subtree.

\[\text{LCA} \Leftrightarrow \text{RMA}\]
Approach 4. Divide & Conquer (Blocks)

- Split array into \( \frac{m}{b} \) blocks of size \( b \)
- Assume \( m \) is divisible by \( b \)
- Precompute minimum in blocks
  - \( k \)th block \( \rightarrow \) \( \text{min}_k \)

Range-min \((i, j)\)

1. Find out in which block \( i, j \) fall
2. Return minimum of:
   - \( \text{min}_i \): \( \text{min} \) in block of \( i \) from \( i \) to the end of the block
   - \( \text{min}_j \): \( \text{min} \) in block of \( j \) from beginning to \( j \)
3. Minimum over minimum of full blocks between \( i \) and \( i \).

Preprocessing time: \( O(n) \) \( \rightarrow \) computing block minimum

Query time: \( O(b) + O\left(\frac{n}{b}\right) = O\left(b + \frac{n}{b}\right) \)

Choose \( b = \sqrt{n} \)

Overall: \( O(\sqrt{n}), O(\sqrt{n}) \) \( \rightarrow \) Approach 4.

Approach 5. Canoical rays

Recall \( O(n^2), O(1) \) "precompute all"

Let's try to precompute fewer queries, so that we can still reconstruct

\( \text{range-min}(i, j) \) is \( O(1) \) time.

We can compute all range-queries of length \( 2^k \)

Precompute range-min \((i, i + 2^k - 1)\)

for all \( i = \frac{m}{2^k} \)

for all \( k = \log_2 m \)

for \( i, k \) such that \( i + 2^k - 1 \leq m \),

\( O(n \log n) \) rays

\[ \text{canonical rays} \]
Claim: preprocessing takes $O(n \log n)$ time.

Use D-P

- first compute all range-runs of length 1
- next compute ___
- ...
- ...
- ... 2
- each takes $O(n)$

Query:

$$\text{range-min}(i,j)$$

- if $j = i + 2^k - 1$, for some $k$ (length of range is power of 2)
  - return precomputed value
- else $j-i$ is between $2^{k-1}$ and $2^{k+1} - 1$ (for some $k$)
  - return $\min\{\text{range-min}(i, i+2^k), \text{range-min}(i+2^k, j)\}$

Overall: $\langle O(n \log n), O(1) \rangle$
Approach 6. Divide & Conquer (Recursively)

\[ \frac{m}{b} \text{ blocks of size } b \]

\[ \text{range-mn}(i,j) \]

need to compute range-mn
1. within blocks \( \rightarrow \text{Micro} \)
2. among block-mins \( \rightarrow \text{Macro} \)

let's use recursively same RMQ solution for both tasks.

**High level scheme**

- Split A into \( \frac{m}{b} \) blocks of size \( b \)
- Compute min in each block \( \rightarrow O(n) \)
- Build RMQ solution within each block with time \( <p_1, q_1> \) \( \rightarrow \text{Micro} \)
- Build RMQ solution between block-mins with time \( <p_2, q_2> \) \( \rightarrow \text{Macro} \)

**Overall preprocessing time:**

\[ p(n) = O(n) + p_1(b) \cdot \frac{m}{b} + p_2 \left( \frac{m}{b} \right) \]

\[ q(n) = q_1(b) + q_2 \left( \frac{m}{b} \right) \]

**Some concrete constructions**

6.a) \( b = 5n \)

\[ \langle p_1, q_1 \rangle = \langle O(n), O(n) \rangle \quad \text{or similar} \]

\[ \langle p_2, q_2 \rangle = \langle O(n), O(n) \rangle \quad \text{or similar} \]

\[ p(n) = O(n) + O(b) \cdot \frac{m}{b} + O(n) = O(n) \]

\[ q(n) = O(n) + O(n) = O(n) \]

\[ \langle O(n), O(n) \rangle \quad \text{just like Approach 4.} \]
6. (a) \( b = \log n \)

\[ \langle p_1, q_1 \rangle = \langle O(n), O(n) \rangle \] "No prep."

\[ \langle p_2, q_2 \rangle = \langle O(n \log n), O(n) \rangle \] "Canonical rays" opp. 5

\[ p(n) = O(n) + \frac{\log n}{\log \log n} + \frac{\log \log n}{\log \log \log n} \leq O(n) \]

\[ q(n) = \frac{O(n \log n)}{\log \log n} = O(\log n) \]

(See runtime time of approach 3)

6. (b) \( b = \log n \)

\[ \langle p_1, q_1 \rangle = \langle O(n \log n), O(n) \rangle \]

\[ \langle p_2, q_2 \rangle = \langle O(n \log n), O(n) \rangle \]

\[ p(n) = O(n) + \frac{\log n}{\log \log n} \cdot \log \log n + \frac{\log \log n}{\log \log \log n} = O(n \cdot \log \log n) \]

\[ q(n) = O(n) \]

[\( \langle O(n \log n), O(1) \rangle \)] best so far!

6. (c) Use canonical rays (opp. 5) for macro

Use augmented tree (opp. 3) for micro

\[ \langle p_1, q_1 \rangle = \langle O(n), O(1) \rangle \]

\[ \langle p_2, q_2 \rangle = \langle O(n \log n), O(1) \rangle \]

\( b = \log n \)

\[ p(n) = O(n) + \frac{\log n}{\log \log n} \cdot \log \log n = O(n) \]

\[ q(n) = O(\log \log n) + O(n) = O(\log \log n) \]

[\( \langle O(n), O(1) \rangle \)]

6. (d) Also "best so far"

6. (e) Phx in recursively

Further improvement

Will not get to \( \langle O(1), O(1) \rangle \)

used are

More idea.
Approach 7: "All-in"

- Block-decomposition, just one level

\[
\frac{n}{b} \quad \text{blocks of size } b
\]

**Micro-structure**

\[
O(\text{log} b), \ O(1)
\]

**Canonical range**

\[
O(b^2), \ O(b)
\]

We need to build RMA structure for \(\frac{n}{b}\) blocks of size \(b\)

- Some blocks may be the same, then we can reuse same RMA structure

\(b=3\)

**RMA**

```
3 | 1 | 2
5 | 1 | 2 | 3
8 | 5 | 2 | 3
1 | 2 | 3
```

\[\text{range-min}(1, 2) = 2\]

**Obs.** If two blocks have the same "order of elements", then we use same RMA

**Obs.** Two blocks are "structurally the same" (for every range-min query, same answer)

\[\text{Cartesian trees of two blocks are the same}\]

Q. How many different Cartesian trees of size \(b\)?

A. \# \(< 4^b\).

**# of different binary trees of size \(n\)**

\[C_n \rightleftharpoons \text{Catalan number}\]

\[
C_n = \sum_{i=1}^{n} C_{i-1} \cdot C_{n-i}
\]

Recurrence:

\[
C_n = \frac{1}{n+1} \cdot \binom{2n}{n} < 4^n
\]

See OEIS.org/A000108 or Wikipedia on "Catalan numbers"
bijection: binary tree $\leftrightarrow$ valid system of parentheses of length $2n$

$\# < 2^{2n} = 4^n$ (number of strings of length $2n$ that encode a valid system of parentheses)

$\#$ of "structurally different blocks" $< 4^b$

Back to approach 7.

- Choose block size $b = \log\sqrt{n}$

- On each block, build Cartesian tree $O(b)$ time, encoding on $\leq 2^b$ bits

- Save type of block for $i = 1, \ldots, \frac{n}{b}$

- $\#$ different types $\leq 4^b = \sqrt{n}$

- For each type of block do $O(n^2)$, $O(n)$ scheme (full preprocessing)

- Total time for this:

  $\sqrt{n} \cdot (\log_4 \sqrt{n})^2 = O(n)$

- Which block -- minimum we use $< \min \{ \log n, O(1) \}$ scheme, we have to preprocess it

  time for this:

  $\frac{n}{b} \cdot \log \frac{n}{b} = \frac{n}{\log \sqrt{n}} \cdot \frac{\log \frac{n}{b}}{\log_4 \sqrt{n}} \leq \frac{n}{\log_4 \sqrt{n}} \cdot \log n = O(n)$

- Total time for all preprocessing: $O(n)$

- Query time:

  $O(n)$ time

  Overall query time: $O(n)$