Range Minimum Query (RMQ)

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

Task: preprocess A, so that we can answer queries

$\text{range-min}(i,j)$, $i \le j$ $\longrightarrow$ return $\min \{A[k] \mid i \le k \le j\}$

(preferably return index k) $\rightarrow$ k s.t. $A[k] < A[t]$ for all $i \le t \le j$

e.g.

1	2	3	4	5
12	3	6	8	17

$\text{range-min}(2,4) \longrightarrow$ 2, $A[2]=3$

$\text{range-min}(3,5) \longrightarrow$ 3, $A[3]=6$

- static* case (array fixed, no insert/delete)

  * some solutions can be made dynamic

- assume array entries are distinct (o.w. perturb by random $[0,\varepsilon]$)

Quality of an algorithm measured as:

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

expressed as $\langle p(n), q(n) \rangle$

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

$O(n)$: must at least read the array

$O(1)$: best possible

(We will get there.)

Observation

If we wanted range-sum, then easy.

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

Idea: precompute all prefix sums S

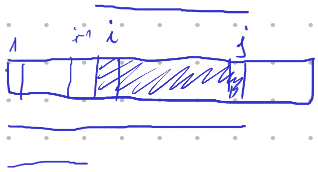
$S[i] = A[1] + A[2] + \ldots + A[i]$

$$\left.\begin{array}{l} S[0] = 0 \\ S[1] = A[1] \\ S[2] = A[1] + A[2] = S[1] + A[2] \\ \ldots \\ S[n] = S[n-1] + A[n] \end{array}\right\} \text{Overall } O(n) \text{ time}$$

range-sum(i,j) $\quad O(1)$

return $\left(S[j] - S[i-1]\right)$

inverse of +



$\langle O(n), O(1)\rangle$

range-min more difficult

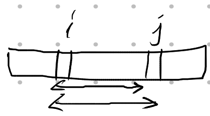
(min no inverse)

Why solve RMQ (Applications)

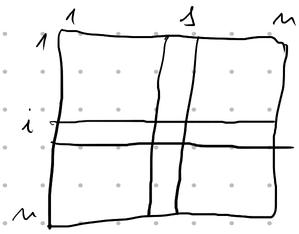
- → string algorithms (pattern matching)
- → algorithms on trees/graphs
- → succinct data structures

Approach 1. Precompute everything

Compute $M[i,j]$ = answer to range-min(i,j)



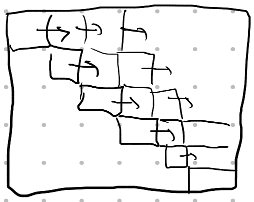
Naively $O(n^3)$



Dynamic Programming $O(n^2)$

$M[i,i] = A[i]$ $\qquad i \le j$

$M[i,j] = \min\{M[i,j-1], A[j]\}$



running time:

$\langle O(n^2), O(1)\rangle$

Approach 1.

range-min(i,j)

| return $M[i,j]$

Approach 2. Precompute nothing

range-min(i,j)

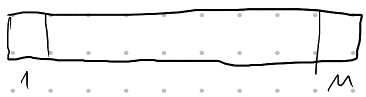
```
m = A[i]
for k = i+1, j
    m = min{m, A[k]}
return m
```

running time:

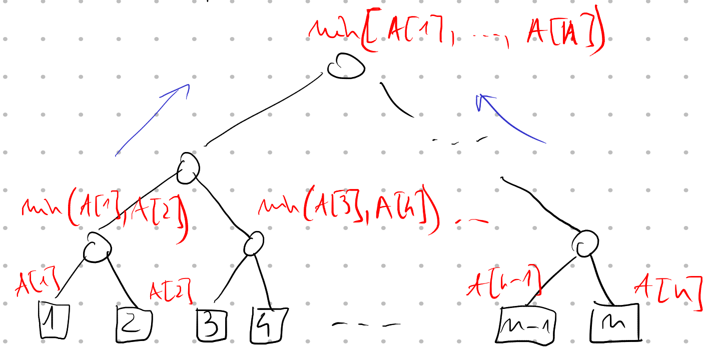
$\langle O(1), O(n)\rangle$

Approach 2.

Approach 3: Augmented trees



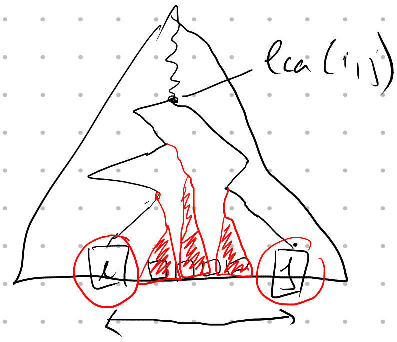
Build a balanced BST, using the indices as keys.



- assume $n = 2^k$
- augment nodes with min array entry among those with index in subtree (also store index of min)

- tree size $2n-1$
- augmentation can be computed in $O(n)$ time (bottom-to-top level by level)

  - answer query range-min(i,j)
    - search for $\boxed{i}$, $\boxed{j}$



- take min over → node-minima for all nodes on path with trees hanging inside
→ and also $\boxed{i}$, $\boxed{j}$ itself

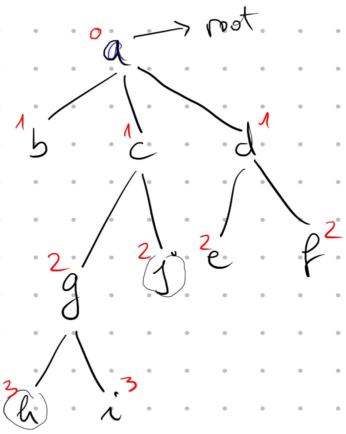
$O(\log n)$ (depth $\le \log_2 n + 1$)

Overall running time $\langle O(n), O(\log n) \rangle$ Approach 3

(this is not yet optimal, we want $\langle O(n), O(1) \rangle$)

Detour: an application of RMQ

Finding lowest common ancestor (lca) in a tree in $O(1)$ time



task: preprocess tree ($O(n)$ size) s.t. lca(i,j) take $O(1)$ time

$\langle O(n), O(1) \rangle$ lca

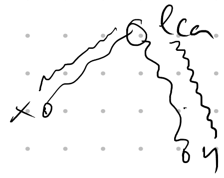
lca$(h,j) \to c$

lca$(g,e) \to a$

→ useful for distance-queries in trees
(store level/depth of nodes)

e.g. XML-documents

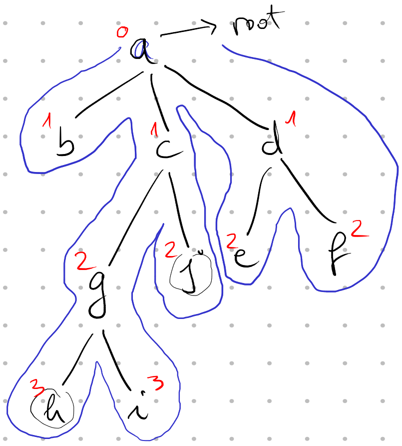
$$\text{dist}(x,y) = \text{depth}(x) + \text{depth}(y) - 2\,\text{depth}(\text{lca}(x,y))$$



Solve LCA problem using RMQ

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

- for every traversed edge, record both endpoints (Euler-tour)



ET: a b a c g h g i g c j c a d e d f d a → size $2n-1$

depths D: 0 1 0 1 2 3 2 3 2 1 2 1 0 1 2 1 2 1 0

$\text{lca}(x,y) \longrightarrow$ return range-min(i_x, i_y)

- range-min: minimum according to depths D
- i_x, i_y: index of some occurrence of x, y in ET

e.g. $\text{lca}(h,j) \to c$

$\text{lca}(g,e) \to a$

$n-1$ edges, each traversed twice

each edge-traversal adds one node + beginning at root $\Longrightarrow 2n-1$ entries

$\langle O(n), O(1)\rangle$ for RMQ $\Longrightarrow$ $\langle O(n), O(1)\rangle$ for LCA (we just showed)

$\overset{?}{\Longleftarrow}$ (next)

Solve RMQ using LCA

Recall: treap or Cartesian tree

tree with nodes (x_i, y_i) $\quad i = 1, \dots, n$

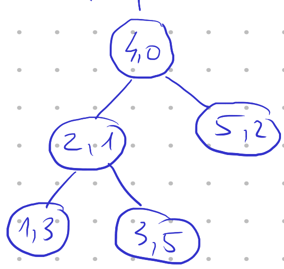
s.t. BST according to x_i values

min-heap according to y_i values

e.g.

x_i, y_i
(1, 3)
(2, 1)
(3, 5)
(4, 0)
(5, 2)

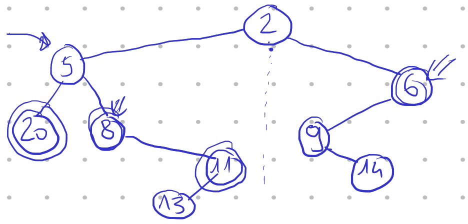
treap



Given array $A[1], \dots, A[n]$ "Cartesian tree of array"

Build a treap on pairs $(i, A[i])$
(Cartesian tree)

i → BST, $A[i]$ → min-heap



A[i]	20	5	8	13	11	2	9	14	6
i	1	2	3	4	5	6	7	8	9

- Given an array, treap can be built in time $O(n)$. (Later)
- Claim: range-min(i,j) is exactly $lca(i,j)$ in treap built over A

It follows that

$$\langle O(n), O(1)\rangle \text{ for lca} \Rightarrow \langle O(n), O(1)\rangle \text{ for RMQ}$$

- build treap
- answer LCA queries on it

<u>Proof</u> (Claim)

(i) $lca(i,j)$ is between i, j

(ii) subtree of $lca(i,j)$ contains entire interval $[i,j]$

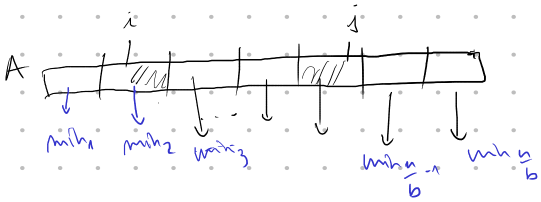
(iii) <u>$lca(i,j)$ has minimum value in all its subtree</u>

→ minimum value $A[i], \dots, A[j]$

$\square$

$$LCA \Leftrightarrow RMQ$$

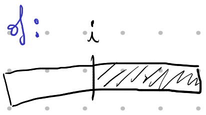
Approach 4. Divide & Conquer (Blocks)



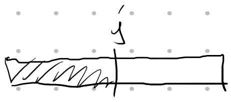
- split array into $\frac{n}{b}$ blocks of size b
- assume n divisible by b
- precompute minima in blocks
  k^{th} block $\longrightarrow$ min_k

range-min (i,j)

- figure out in which block i,j fall
- return minimum of:

(1) 

min in block of i from i to the end of the block

(2) 

min in block of j from beginning to j

(3) minimum over minima of full blocks between i and j.

(if i,j fall in the same block, then easier, then we look in that block only:

)

- preprocessing time: $O(n)$ $\longrightarrow$ computing block minima
- query time: $\underbrace{O(b)}_{\text{inside block}} + \underbrace{O\left(\frac{n}{b}\right)}_{\text{between blocks}} = O\left(b + \frac{n}{b}\right)$

choose $b = \sqrt{n}$

Query time $O(\sqrt{n})$

Overall: $\langle O(n), O(\sqrt{n}) \rangle$ Approach 4.

Approach 5. Canonical ranges.

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

Let's try to precompute fewer queries, so that we can still reconstruct range-min(i,j) in $O(1)$ time.

Idea: compute all range-queries of length 2^k $\longrightarrow$ Canonical ranges

precompute range-min$(i, i+2^k-1)$

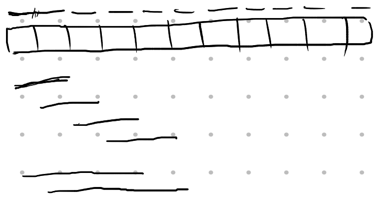
for all $i = \overline{1,n}$

for all $k = \overline{0, \log n}$

$\Big\}$ for i,k such that $i+2^k-1 \leq n$

$O(n \log n)$ ranges.

Claim: preprocessing takes $O(n \log n)$ time.



Use D-P

- first compute all ranges of length 1
- next compute — — 2
- — 4
- ...
- 2^k

each takes $O(1)$

(Computing a range-min for a range of size 2^k can be done as min of two (already computed) of size 2^{k-1})

Queries

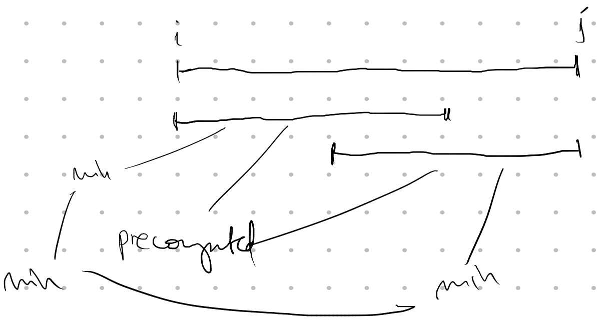
range-min (i,j)

- if $j = i + 2^k - 1$, for some k. (length of range is power of 2 r-min is precomputed)

  return precomputed value.

- else $j-i$ is between $2^k - 1$ and $2^{k+1} - 1$ (for some k)

  return $\min\{\text{range-min}(i, i+2^k-1), \text{range-min}(j-2^k+1, j)\}$.



$O(1)$ time

Interval of interest is covered by two intervals of size 2^k (minimum is correct, even if overlap)

e.g.

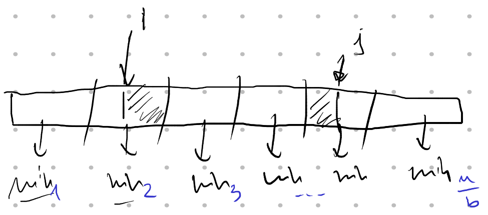
length 2^4, so precomputed

range-min$(3,20) \longrightarrow$ min(range-min$(3,18)$, range-min$(5,20)$)

Approach 5.

Overall: $\langle O(n\log n), O(1)\rangle$

Approach 6. Divide & Conquer (Recursively)



$\frac{n}{b}$ blocks of size b

range-min(i,j)

need to compute range-min (1) within blocks → MICRO

(2) among block-minima → MACRO

Let's use recursively some RMQ solution for both tasks.

High level scheme

- Split A into $\frac{n}{b}$ blocks of size b
- Compute min in each block $\longrightarrow O(n)$
- Build RMQ solution within each block with time $\langle p_1, q_1\rangle$ — MICRO
- Build RMQ solution between block-minima with time $\langle p_2, q_2\rangle$ — MACRO

Overall preprocessing time:

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

$$q(n) = q_1(b) + q_2\left(\frac{n}{b}\right)$$

Some concrete constructions

6.a) $b = \sqrt{n}$

$$\left.\begin{aligned}\langle p_1, q_1\rangle &= \langle O(1), O(n)\rangle\\ \langle p_2, q_2\rangle &= \langle O(1), O(n)\rangle\end{aligned}\right\}$$ "no preproc." Approach 2

$$p(n) = O(n) + O(1)\cdot\frac{n}{b} + O(1) = O(n)$$

$$q(n) = O(\sqrt{n}) + O(\sqrt{n}) = O(\sqrt{n})$$

6.a) $\langle O(n), O(\sqrt{n})\rangle$

Just like Approach 4.

6.b) $b = \log n$

$\langle p_1, q_1 \rangle = \langle O(1), O(n) \rangle$ "no prepr."

$\langle p_2, q_2 \rangle = \langle O(n\log n), O(1) \rangle$ "Canonical ranges" appr. 5

$$p(n) = O(n) + \frac{n}{\log n} + \frac{n}{\log n} \cdot \log\left(\frac{n}{\log n}\right) \leq O(n)$$

$$q(n) = O(\log n) + O(1) = O(\log n)$$

6. b) $\langle O(n), O(\log n) \rangle$

(same running time as approach 3)

6. c) $b = \log n$

$\langle p_1, q_1 \rangle = \langle O(n\log n), O(1) \rangle$

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

$$p(n) = O(n) + \frac{n}{\cancel{\log n}} \cdot \cancel{\log n} \cdot \log\log n + \underbrace{\frac{n}{\log n} \cdot \log\left(\frac{n}{\log n}\right)}_{O(n)} = O(n \cdot \log\log n)$$

$$q(n) = O(1)$$

6.c) $\langle O(n \log\log n), O(1) \rangle$ best so far!

6. d) use canonical ranges (appr. 5) for macro

use augmented tree (appr. 3) for micro

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

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

$b = \log n$

$$p(n) = O(n) + \frac{n}{b} + \frac{n}{\log n} \cdot \log\left(\frac{n}{\log n}\right) = O(n)$$

$$q(n) = O(\log\log n) + O(1) = O(\log\log n)$$

Bottomline: $\langle O(n), O(\log\log n) \rangle$ 6.d)

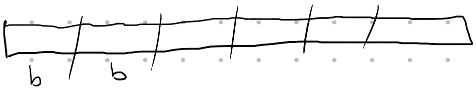
also "best so far"

6. e) Plug in recursively

$\longrightarrow$ Further improvement will not get to $\langle O(n), O(1) \rangle$ $\rightarrow$ need one more idea.

Approach 7. "All-In"

- Block-decomposition, just one level



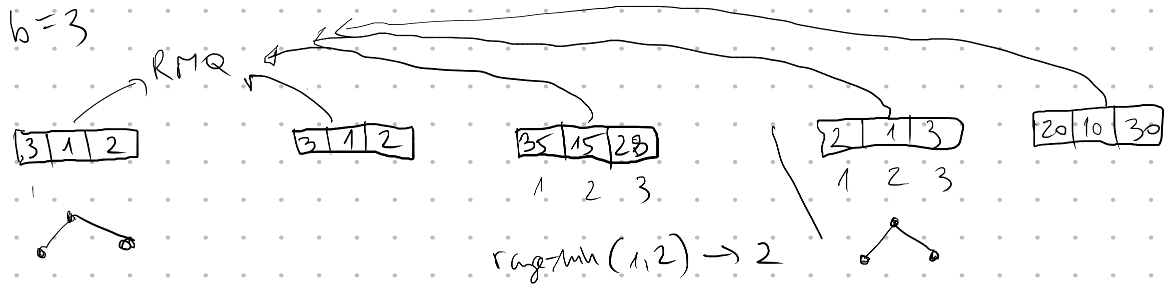
$\frac{n}{b}$ blocks of size b

macro-structure $\langle O(n\log n), O(1)\rangle$ (canonical ranges approach 5.)

micro-structure $\longrightarrow \langle O(n^2), O(1)\rangle$ (precompute all, but take advantage of blocks that are the same)

we need to build RMQ structure for $\frac{n}{b}$ blocks of size b

- some blocks may be the same, then we can reuse same RMQ structure

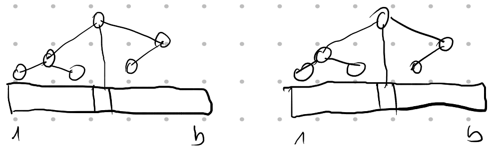


Obs. If two blocks have the same "ordering of elements", then use same RMQ micro-structure

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

$\Updownarrow$

Cartesian trees of two blocks are the same



recall LCA, RMQ equivalence. If Cartesian trees the same, LCA gives same $\Rightarrow$ RMQ same

Q. How many different Cartesian trees of size b?

A. $\# < 4^b$.

\# of different binary trees of size n is $C_n \longrightarrow$ Catalan number.

n	1	2	3	4	5	
C_n	1	2	5	14	42	...

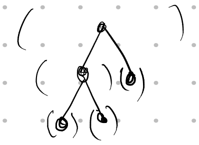
See OEIS.org/A000108 or Wikipedia on "Catalan numbers"

Recurrence:

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

base case: $C_0 = 1$.

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



bijection binary trees $\longleftrightarrow$ valid systems of parentheses
of size n / of length $2n$

$$\underline{((()())())} \quad \text{length } 2n$$

$$\# < 2^{2n} = \underline{4^n}$$

(# strings of length $2n$ that encode a valid system of parentheses)

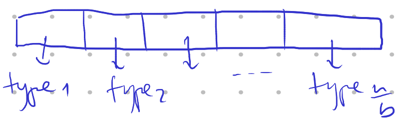
of "structurally different blocks" $< 4^b$

Back to approach 7.

Preprocessing

- choose block size $b = \log_4 \sqrt{n}$
- on each block build Cartesian tree $O(b)$ time, encoding on $\leq 2b$ bits

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



$1 \leq type_i \leq \sqrt{n}$

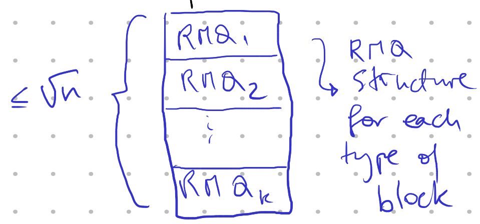
different types $< 4^b = \sqrt{n}$

(time for this: $O(n)$)

- For each type of block do $\langle O(n^2), O(1) \rangle$ scheme (full preprocessing)

  (total time for this:

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



- within block-minima we use $\langle n \log n, O(1) \rangle$ scheme, we have to preprocess it

  time for this:

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

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

- Query-time

(type of each block saved in preprocessing)

Overall query time: $O(1)$

Finally: $\langle O(n), O(1) \rangle$ Approach 7.