9 Range-Minimum Queries

27 April 2020

Sebastian Wild

version 2020-04-28 14:54

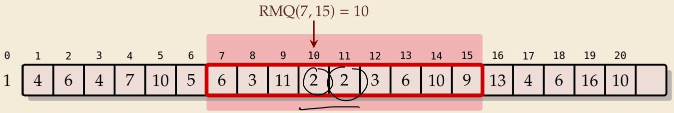
Outline

9 Range-Minimum Queries

9.1 Introduction

Range-minimum queries (RMQ)

- **Given:** Static array $A[0..n)$ of numbers
  (array/numbers don't change)
- **Goal:** Find minimum in a range;
  A known in advance and can be preprocessed



- **Nitpicks:**
  - Report *index* of minimum, not its value
  - Report *leftmost* position in case of ties

Clicker Question

Given the array from the slides, what is $\mathrm{RMQ}_A(1,6)$ = 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	4	6	4	7	10	5	6	3	11	2	2	3	6	10	9	13	4	6	16	10

pingo.upb.de/622222

Rules of the Game

- comparison-based $\rightsquigarrow$ values don't matter, only relative order
- Two main quantities of interest:
  1. **Preprocessing time**: Running time $P(n)$ of the preprocessing step
  2. **Query time**: Running time $Q(n)$ of one query (using precomputed data)
- Write "$\langle P(n), Q(n) \rangle$ time solution" for short

prep- query

(also: space usage $\le P(n)$)

Clicker Question

What do you think, what running times can we achieve? For a $\langle P(n), Q(n) \rangle$ time solution, enter "<P(n),Q(n)>".

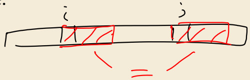
pingo.upb.de/622222

9.2 RMQ, LCP, LCE, LCA — WTF?

Application 4: Longest Common Extensions

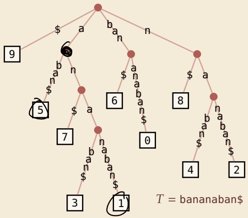
- We implicitly used a special case of a more general, versatile idea:

  The ***longest common extension (LCE)*** data structure:

  - **Given:** String $T[0..n-1]$
  - **Goal:** Answer LCE queries, i. e.,
    given positions i, j in T,
    how far can we read the same text from there?
    formally: $\text{LCE}(i, j) = \max\{\ell : T[i..i+\ell) = T[j..j+\ell)\}$

$\rightsquigarrow$ use suffix tree of T!



- In $\mathcal{T}$: $\text{LCE}(i, j) = \text{LCP}(T_i, T_j)$ $\rightsquigarrow$ same thing, different name!
  (longest common prefix of ith and jth suffix)
  $=$ string depth of *lowest common ancester (LCA)* of leaves $\boxed{i}$ and $\boxed{j}$

- in short: $\text{LCE}(i, j) = \text{LCP}(T_i, T_j) = \text{stringDepth}\big(\text{LCA}(\boxed{i}, \boxed{j})\big)$

18

Recall Unit 6

Efficient LCA

How to find lowest common ancestors?

- Could walk up the tree to find LCA ⇝ $\Theta(n)$ worst case
- Could store all LCAs in big table ⇝ $\Theta(n^2)$ space and preprocessing

Amazing result: Can compute data structure in $\Theta(n)$ time and space that finds any LCA is **constant(!) time**.

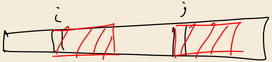
- a bit tricky to understand
- but a theoretical breakthrough
- and useful in practice

and suffix tree construction inside . . .

⇝ for now, use $O(1)$ LCA as black box.

⇝ After linear preprocessing (time & space), we can find LCEs in $O(1)$ time.

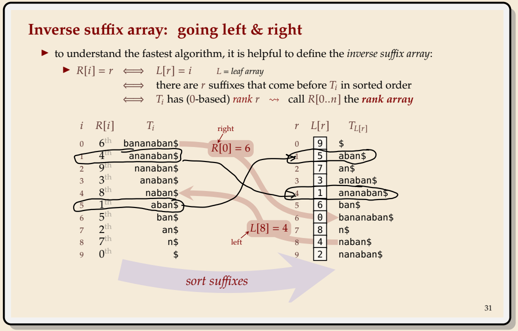
Finally: Longest common extensions

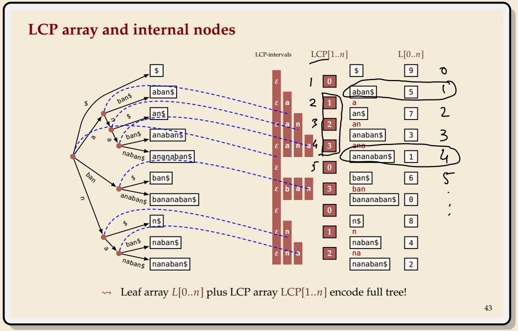


- In Unit 6: Left question open how to compute LCA in suffix trees
- But: Enhanced Suffix Array makes life easier!

$$\mathrm{LCE}(i,j) \;=\; \mathrm{RMQ}_{\mathrm{LCP}}(R[i]+1, R[j])$$

LCP[↓] = LCP[2] = 1 $\mathrm{RMQ}_{\mathrm{LCP}}(2,4) = 2$





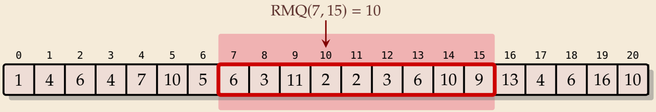
RMQ Implications for LCE

- Recall: Can compute (inverse) suffix array and LCP array in $O(n)$ time

$\rightsquigarrow$ A $\langle P(n), Q(n) \rangle$ time RMQ data structure implies a $\langle P(n), Q(n) \rangle$ time solution for longest-common extensions

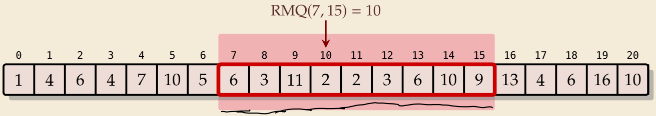
9.3 Sparse Tables

Trivial Solutions



- Two easy solutions show extreme ends of scale:

Trivial Solutions



- Two easy solutions show extreme ends of scale:

1. **Scan on demand**

  - no preprocessing at all
  - answer RMQ(i, j) by scanning through $A[i..j]$, keeping track of min

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

Trivial Solutions

$RMQ(7, 15) = 10$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	4	6	4	7	10	5	6	3	11	2	2	3	6	10	9	13	4	6	16	10

- Two easy solutions show extreme ends of scale:

1. **Scan on demand**
   - no preprocessing at all
   - answer $RMQ(i, j)$ by scanning through $A[i..j]$, keeping track of min

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

2. **Precompute all**
   - Precompute all answers in a big 2D array $M[0..n)[0..n)$
   - queries simple: $RMQ(i, j) = M[i][j]$

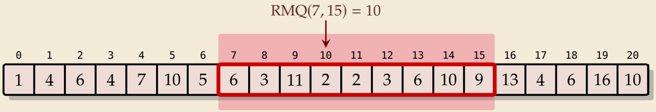
   $\rightsquigarrow \langle O(n^3), O(1) \rangle$

$0 \le i < n$

$i \le j < n$

$\Theta(n^2)$ entries

Trivial Solutions



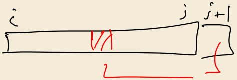
- Two easy solutions show extreme ends of scale:

1. **Scan on demand**
   - no preprocessing at all
   - answer $\mathrm{RMQ}(i,j)$ by scanning through $A[i..j]$, keeping track of min

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

2. **Precompute all**
   - Precompute all answers in a big 2D array $M[0..n)[0..n)$
   - queries simple: $\mathrm{RMQ}(i,j) = M[i][j]$

   $\rightsquigarrow$ $\langle O(n^3), O(1) \rangle$
   - Preprocessing can reuse partial results $\rightsquigarrow$ $\langle O(n^2), O(1) \rangle$



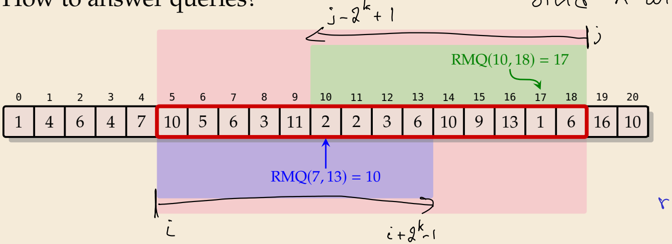
Sparse Table

- **Idea:** Like "precompute-all", but keep only some entries
- store $M[i][j]$ iff $\ell = j - i + 1$ is 2^k. $0 \le i < n$
  - $\rightsquigarrow \le n \cdot \lg n$ entries ↪ store $M[i][k]$

Sparse Table

- **Idea:** Like "precompute-all", but keep only some entries
- store $M[i][j]$ iff $\ell = j - i + 1$ is 2^k.
  $\rightsquigarrow \le n \cdot \lg n$ entries
- How to answer queries?

$\ell = j - i + 1$: Can always find k with: $\frac{\ell}{2} \le 2^k \le \ell$



$\leadsto RMQ(i,j)$

$= \arg\min\{A[rmq], A[rmq]\}$

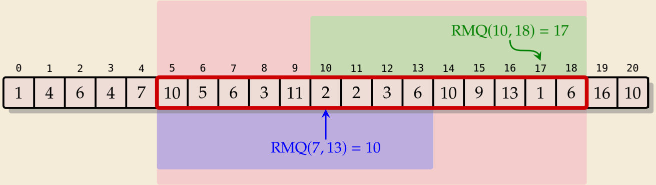
$rmq = RMQ(i, i+2^k-1)$

$rmq = RMQ(j-2^k+1, j)$

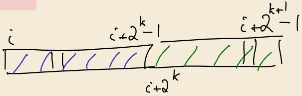
Sparse Table

- **Idea:** Like "precompute-all", but keep only some entries
- store $M[i][j]$ iff $\ell = j - i + 1$ is 2^k.
  $\rightsquigarrow \le n \cdot \lg n$ entries

- How to answer queries?



- Preprocessing can be done in $O(n \log n)$ times



$\rightsquigarrow \langle O(n \log n), O(1) \rangle$ time solution!

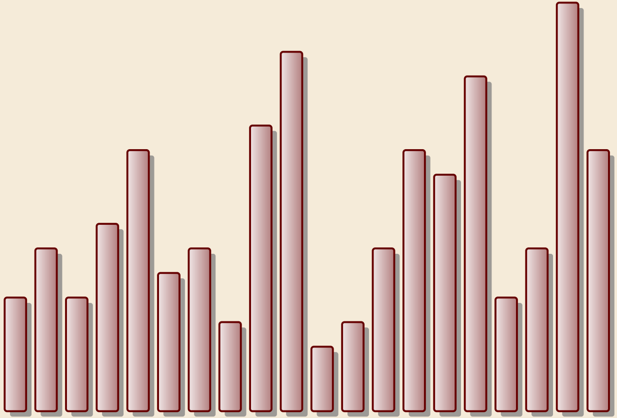
eventually $\langle O(n), O(1) \rangle$

9.4 Cartesian Trees

Range-maximum queries

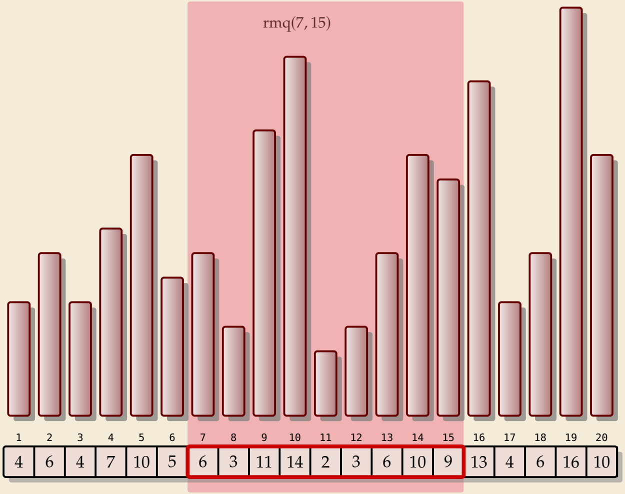
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
4	6	4	7	10	5	6	3	11	14	2	3	6	10	9	13	4	6	16	10

Range-maximum queries



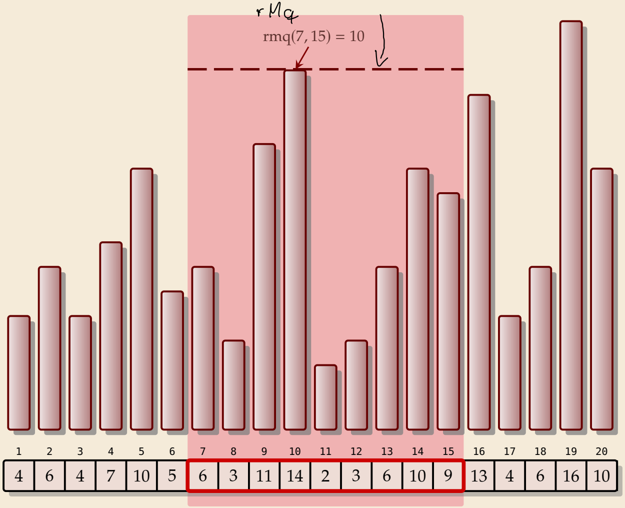
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
4	6	4	7	10	5	6	3	11	14	2	3	6	10	9	13	4	6	16	10

Range-maximum queries



- **R**ange-**m**ax **q**ueries on array A:
  $\text{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  $= \textit{index}$ of max

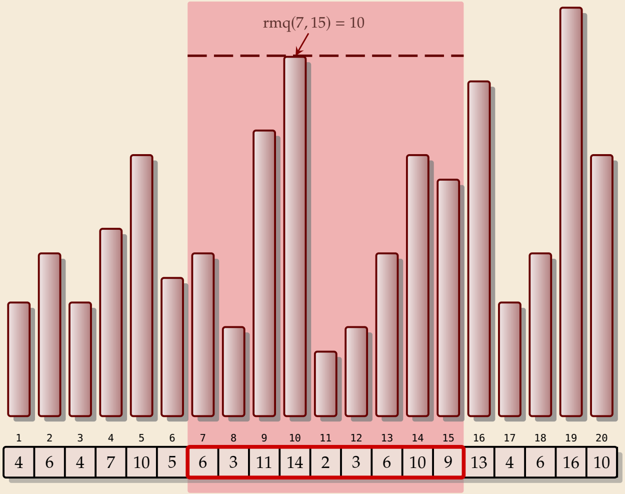
Range-maximum queries



- **Range-max queries** on array A:

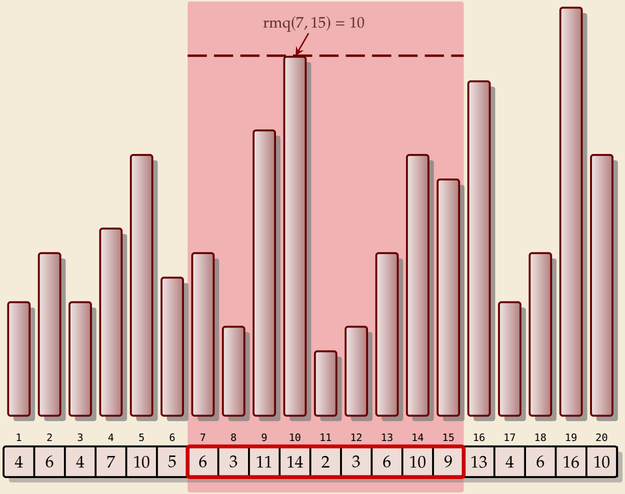
$$\text{rmq}_A(i,j) = \arg\max_{i \le k \le j} A[k]$$
$$= \textit{index} \text{ of max}$$

Range-maximum queries



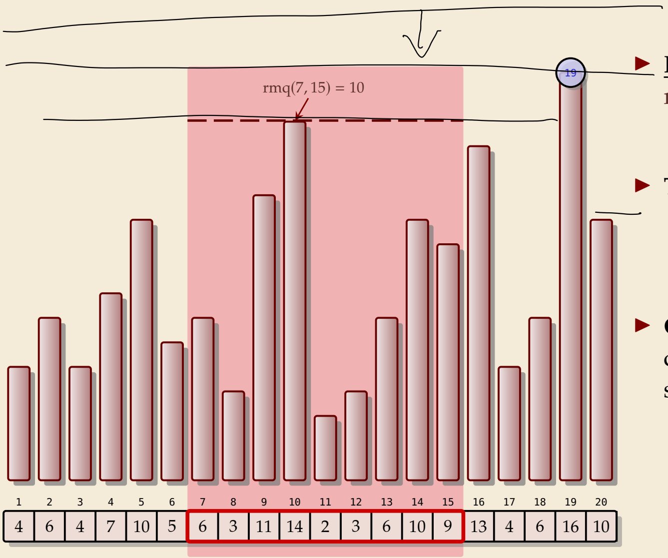
- **Range-max queries** on array A:
  $$\text{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$$
  $$= \textit{index} \text{ of max}$$
- **Task:** Preprocess A, then answer RMQs fast

Range-maximum queries



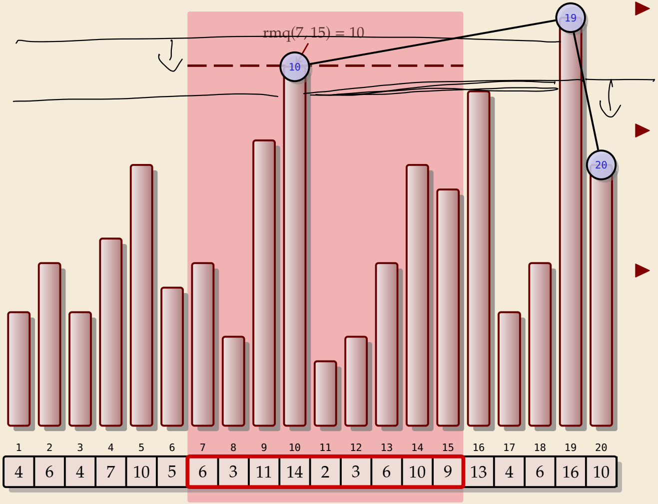
- **Range-max queries** on array A:
  $\mathrm{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  $= \textit{index}$ of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!

Range-maximum queries



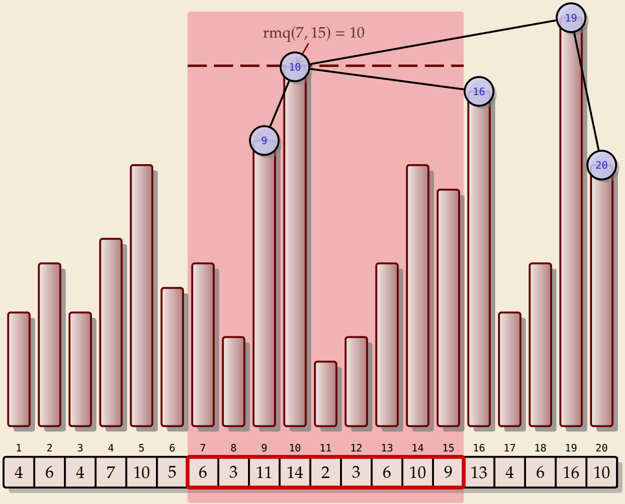
- **Range-max queries** on array A:
  $\text{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  = *index* of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!
- **Cartesian tree:** (cf. *treap*)
  construct binary tree by
  sweeping line down

Range-maximum queries



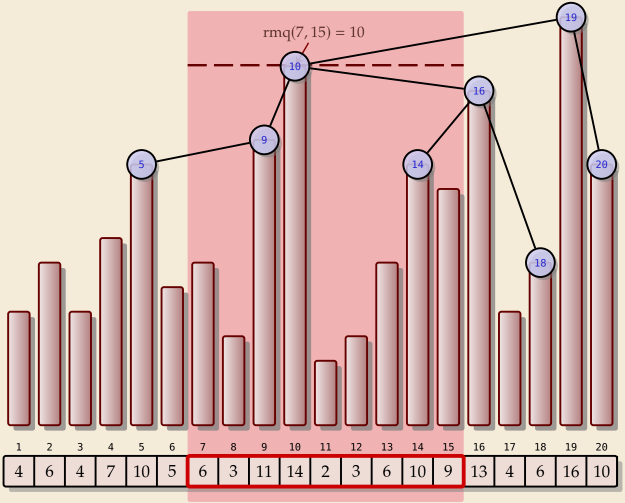
- **Range-max queries** on array A:
  $\text{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  $= \textit{index}$ of max
- **Task:** Preprocess A, then answer RMQs fast ideally constant time!
- **Cartesian tree:** (cf. *treap*) construct binary tree by sweeping line down

Range-maximum queries



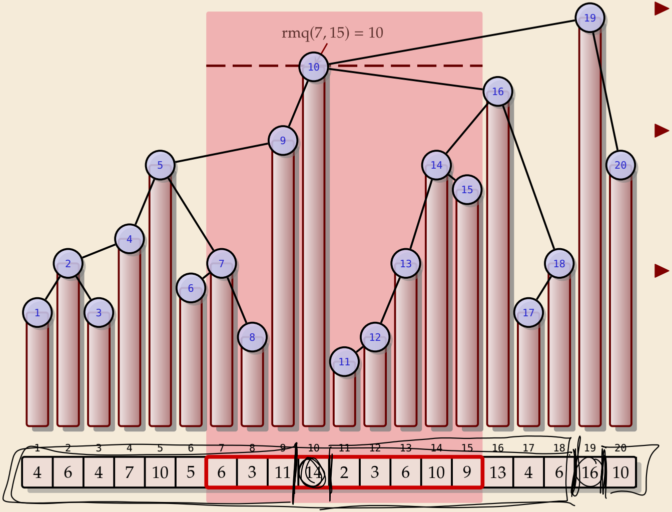
- **Range-max queries** on array A:
  $\text{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  $= \textit{index}$ of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!
- **Cartesian tree:** (cf. *treap*)
  construct binary tree by
  sweeping line down

Range-maximum queries



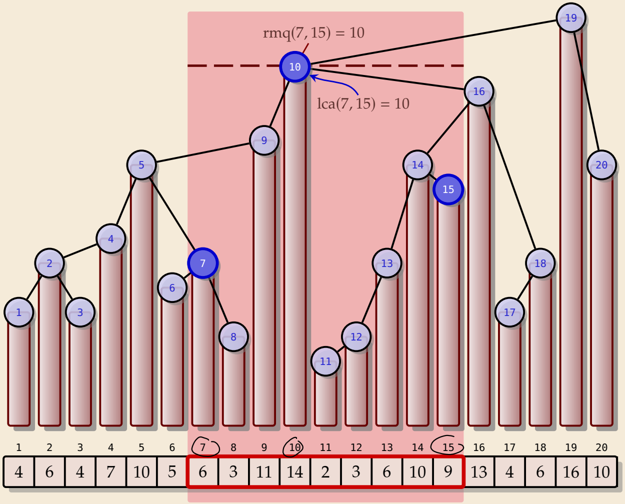
- **Range-max queries** on array A:
  $\text{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  $= \textit{index}$ of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!
- **Cartesian tree:** (cf. *treap*)
  construct binary tree by
  sweeping line down

Range-maximum queries



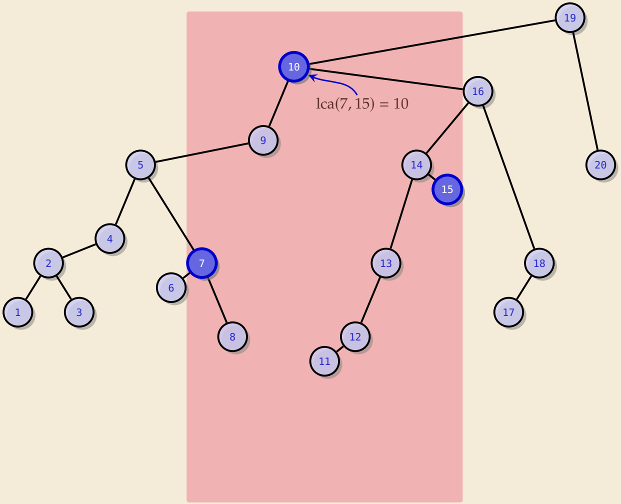
- **R**ange-**m**ax **q**ueries on array A:
  $\text{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  $= \textit{index}$ of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!
- **Cartesian tree:** (cf. *treap*)
  construct binary tree by
  sweeping line down

Range-maximum queries



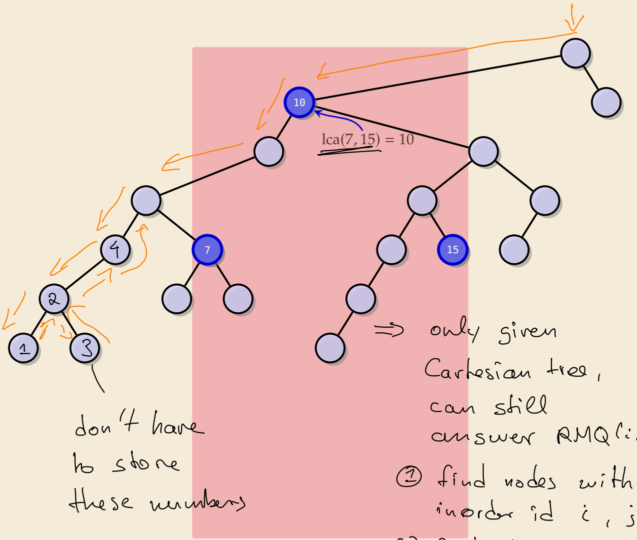
- **Range-max queries** on array A:
  $\mathrm{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  $= \mathit{index}$ of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!
- **Cartesian tree:** (cf. *treap*)
  construct binary tree by
  sweeping line down
- $\mathrm{rmq}(i, j) =$
  lowest common ancestor (LCA)

Range-maximum queries



- **Range-max queries** on array A:
  $\text{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  $= \textit{index}$ of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!
- **Cartesian tree:** (cf. *treap*)
  construct binary tree by
  sweeping line down
- $\text{rmq}(i, j) =$
  lowest common ancestor (LCA)

Range-maximum queries



⇒ only given Cartesian tree, can still answer RMQ(i, j)

don't have to store these numbers

① find nodes with inorder id i, j

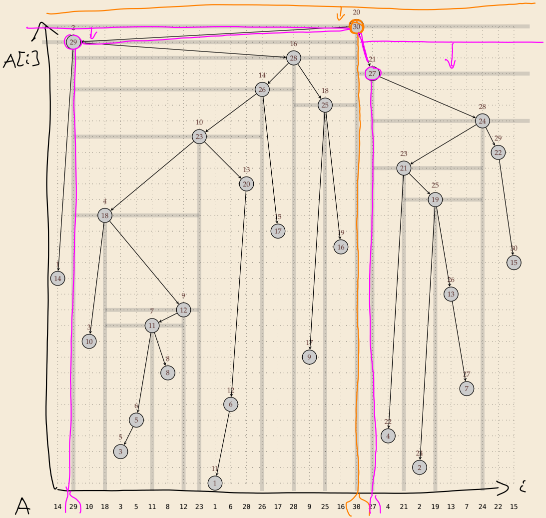
② find their LCA

- **Range-max queries** on array A:
  $\text{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  $= \textit{index}$ of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!

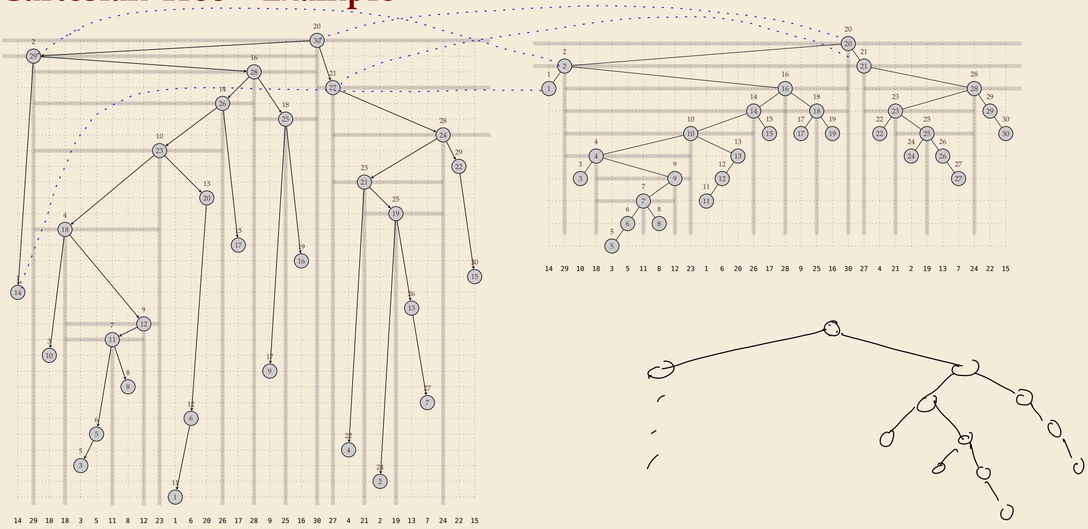
- **Cartesian tree:** (cf. *treap*)
  construct binary tree by
  sweeping line down
- $\text{rmq}(i, j)$ = inorder of
  lowest common ancestor (LCA)
  of ith and jth node in inorder

③ return inorder index of LCA

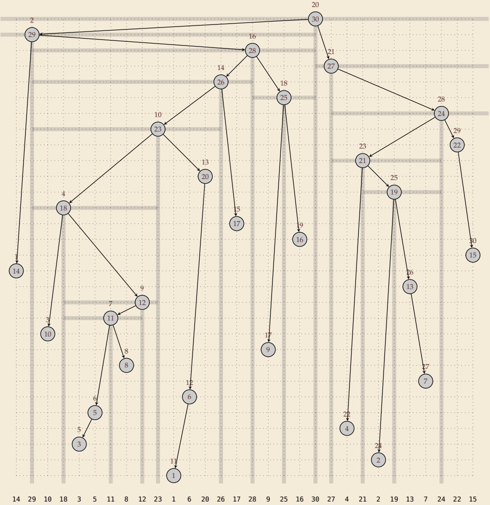
Cartesian Tree – Example

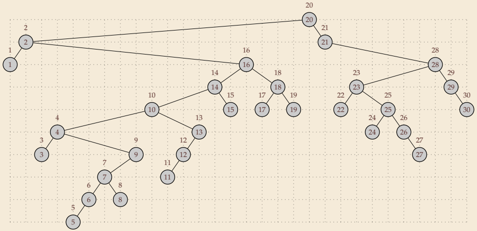


Cartesian Tree – Example



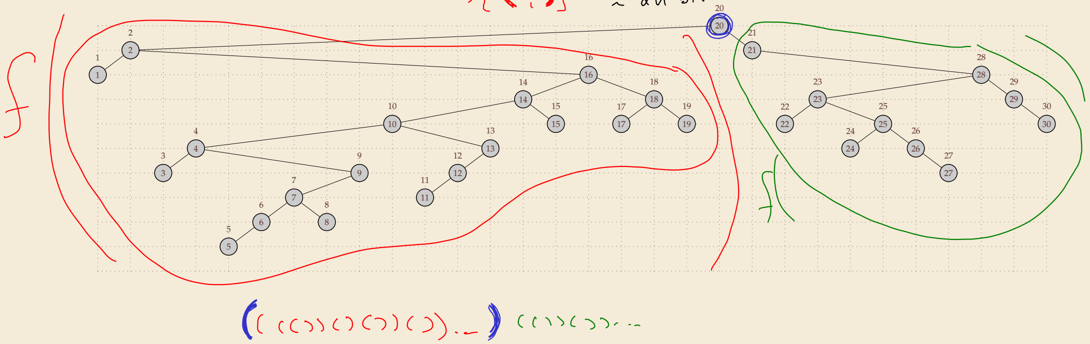
Cartesian Tree – Example

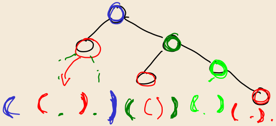


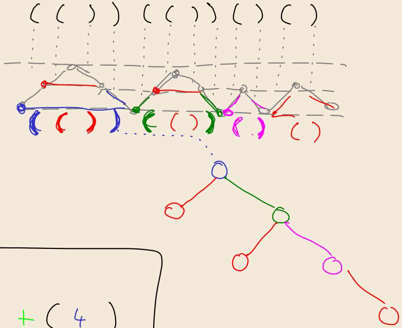


Counting binary trees

- all RMQ answers are determined by Cartesian tree
- How many different Cartesian trees are there for $A[0..n)$?
  - known result: Catalan numbers $\frac{1}{n+1}\binom{2n}{n}$
  - easy to see: $\le 2^{2n}$







Observation: ()-strings is "balanced"

((5)) + ((7)) + (3) + (4)

(valid expression

"(,)" ← 1

") (" ← +

) ( ))) $\not\Rightarrow$ tree

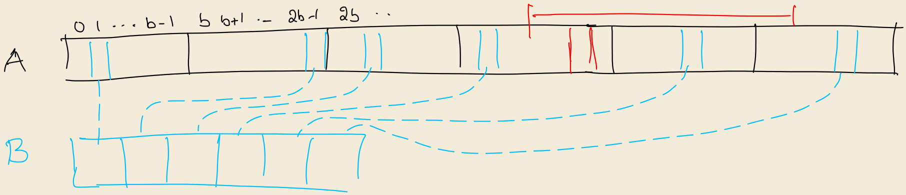
binary tree with n nodes can be encoded as 2n-bit string.

9.5 “Four Russians” Table

Bootstrapping

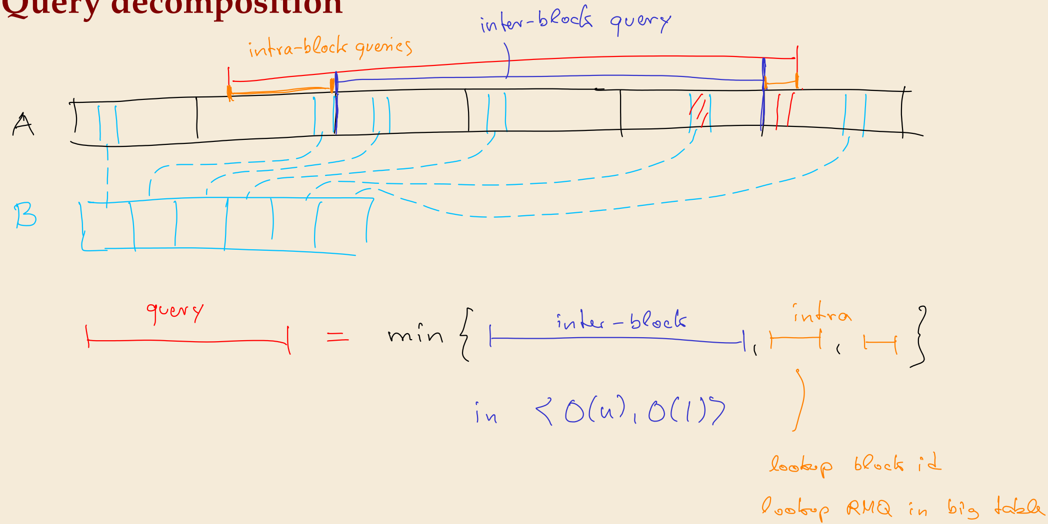
- We know a $\langle O(n \log n), O(1) \rangle$ time solution (sparse table)
- If we use that for $m = \Theta(n/\log n)$ elements, $O(m \log m) = O(n)$!

- Break A into blocks of $b = \lceil \frac{1}{4} \lg n \rceil$ numbers
- Create array of block minima $B[0..m]$ for $m = \lceil n/b \rceil = O(n/\log n)$
  - ⇝ Use sparse tables for B.



$\Rightarrow$ can find $RMQ_B(\cdot,\cdot)$ in $\langle O(n), O(1) \rangle$

Query decomposition



Precomputing intra-block queries

It remains to solve intra-block queries

want $\langle O(n), O(1) \rangle$ time overall

(preprocessing for all $\lceil \frac{n}{b} \rceil = \Theta\left(\frac{n}{\log n}\right)$ blocks

"Four Russians": many blocks, but all just $b = \lceil \frac{1}{4} \lg n \rceil$ numbers

$\rightarrow$ Cartesian trees of b elements (1 block)

can be encoded using $2b$ bits $\leq \frac{1}{2} \lg n$

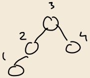
$\Rightarrow$ number of different Cartesian trees wrt RMQ is $\leq 2^{2b} = (2^{\lg n})^{1/2} = \sqrt{n}$

$\Rightarrow$ many equivalent blocks

① for each block we compute & store its <u>type</u> $O(n)$

″ {} of Cartesian tree
≙ binary repr. of number in $\{0, \dots, \sqrt{n}\}$

② compute a big lookup table of all RMQ answers for all types



block ID	i	j	RMQ(i,j)
⋮			
((()))()	1	2	2
((()))()	1	3	3
″	1	4	3
″	2	3	3
″	2	4	3
⋮			

$\sqrt{n} \cdot b^2$ rows

$= \Theta(\sqrt{n} \log^2(n))$ rows

total: preprocessing

1. block types
2. lookup table
3. bootstrap ds for B

} $O(n)$

query: $O(1)$

Discussion

- $\langle O(n), O(1) \rangle$ time solution for RMQ

$\rightsquigarrow$ $\langle O(n), O(1) \rangle$ time solution for LCE in strings!

👍 optimal preprocessing and query time!

👎 a bit complicated

Research questions:

- Reduce the space usage
- Avoid access to A at query time