7 Googling Genomes

15 January 2026

Prof. Dr. Sebastian Wild

CS594 (Winter 2025-26)
Philipps-Universität Marburg
version `2026-01-15 19:01 H`

Outline

7 Googling Genomes

Recall Unit 6

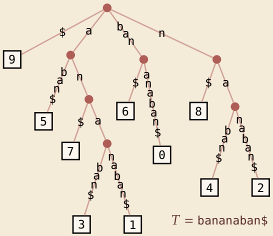
Application 4: Longest Common Extensions

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

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

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

$\rightsquigarrow$ use suffix tree of T!



(length of) longest common prefix
of ith and jth suffix

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

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

Recall Unit 6

Efficient LCA

How to find lowest common ancestors?

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

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

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

and suffix tree construction inside . . .

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

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

7.1 Range-Minimum Queries

Range-minimum queries (RMQ)

array/numbers don't change

- **Given:** Static array $A[0..n)$ of numbers
- **Goal:** Find minimum in a range;
  A known in advance and can be preprocessed

$\text{RMQ}(7, 15) = 10$

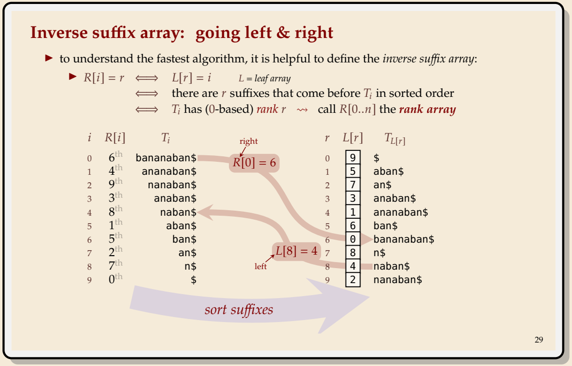
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	4	6	4	7	10	5	6	3	11	2	2	3	6	10	9	13	4	6	16	10

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

Finally: Longest common extensions

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

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

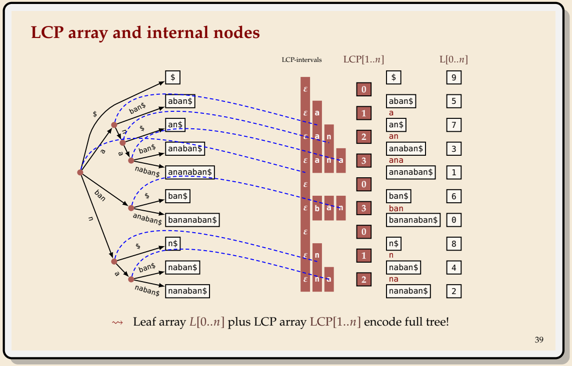


Inverse suffix array: going left & right

- to understand the fastest algorithm, it is helpful to define the *inverse suffix array*:
  - $R[i] = r \iff L[r] = i$ L = *leaf array*
    $\iff$ there are r suffixes that come before T_i in sorted order
    $\iff$ T_i has (0-based) *rank* r $\rightsquigarrow$ call $R[0..n]$ the ***rank array***

i	$R[i]$	T_i
0	6th	bananaban$
1	4th	ananaban$
2	9th	nanaban$
3	3th	anaban$
4	8th	naban$
5	1th	aban$
6	5th	ban$
7	2th	an$
8	7th	n$
9	0th	$

r	$L[r]$	$T_{L[r]}$
0	9	$
1	5	aban$
2	7	an$
3	3	anaban$
4	1	ananaban$
5	6	ban$
6	0	bananaban$
7	8	n$
8	4	naban$
9	2	nanaban$



LCP array and internal nodes

$\rightsquigarrow$ Leaf array $L[0..n]$ plus LCP array $\mathrm{LCP}[1..n]$ encode full tree!

39

Rules of the Game

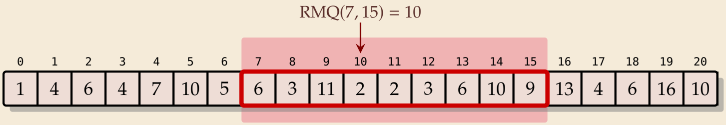
- For the following, consider RMQ on arbitrary arrays
- comparison-based $\rightsquigarrow$ values don't matter, only relative order
- Two main quantities of interest:
  1. **Preprocessing time**: Running time $P(n)$ of the preprocessing step $\rightsquigarrow$ space usage $\le P(n)$
  2. **Query time**: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

RMQ Implications for LCE

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

$\rightsquigarrow$ $\langle P(n), Q(n) \rangle$ time RMQ data structure implies
$\langle P(n) + O(n), Q(n) \rangle$ time LCE data structure

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: $\mathrm{RMQ}(i, j) = M[i][j]$

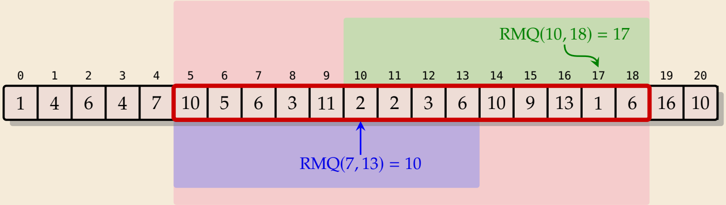
   $\rightsquigarrow$ $\langle O(n^3), O(1) \rangle$

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

7.2 RMQ – Sparse Table Solution

Sparse Table

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



1. Find k with $\ell/2 \le 2^k \le \ell$
2. Cover range $[i..j]$ by
   2^k positions right from i and
   2^k positions left from j
3. $\mathrm{RMQ}(i, j) =$
   $\arg\min\{A[rmq_1], A[rmq_2]\}$

   with $rmq_1 = \mathrm{RMQ}(i, i + 2^k - 1)$
   $rmq_2 = \mathrm{RMQ}(j - 2^k + 1, j)$

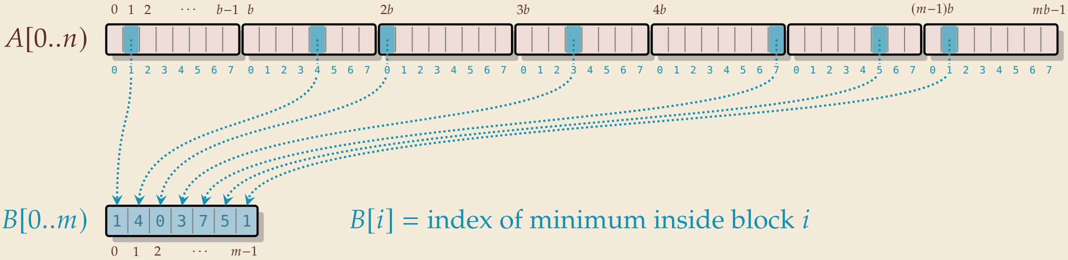
- Preprocessing can be done in $O(n \log n)$ times

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

Bootstrapping

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

- Break A into blocks of $b = O(\log n)$ numbers
- Create array of block minima $B[0..m)$ for $m = \lceil n/b \rceil = O(n/\log n)$

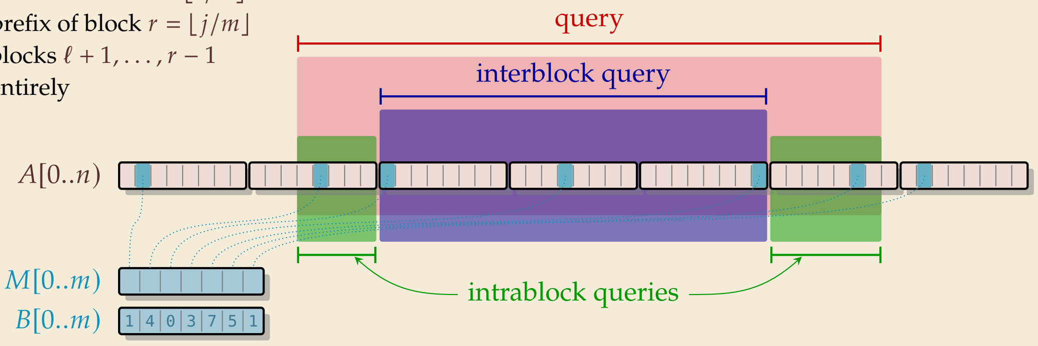


$\rightsquigarrow$ Use sparse table solution for B.

$\rightsquigarrow$ Can solve RMQs in $B[0..m)$ in $\langle O(n), O(1) \rangle$ time

Query decomposition

- Query $\mathrm{RMQ}_A(i,j)$ covers
  - suffix of block $\ell = \lfloor i/m \rfloor$
  - prefix of block $r = \lfloor j/m \rfloor$
  - blocks $\ell+1,\ldots,r-1$ entirely

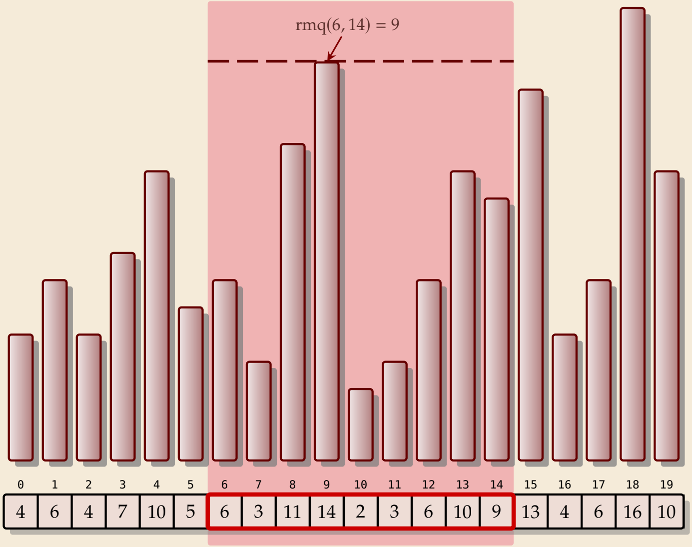


- $\mathrm{RMQ}_A(i,j) \;=\; \underset{k\in K}{\arg\min}\, A[k] \quad \text{with} \;\; K \;=\; \left\{ \begin{array}{l} \mathrm{RMQ}_{\text{block } \ell}\big(i-\ell b,\ (\ell+1)b-1\big), \\ b \cdot \mathrm{RMQ}_M\big(\ell+1, r-1\big) + \\ \qquad B\big[\mathrm{RMQ}_M\big(\ell+1, r-1\big)\big], \\ \mathrm{RMQ}_{\text{block } r}\big(rb,\ j-rb\big) \end{array} \right\}$

$\rightsquigarrow$ only 3 possible values to check
if intrablock and interblock queries known ✓

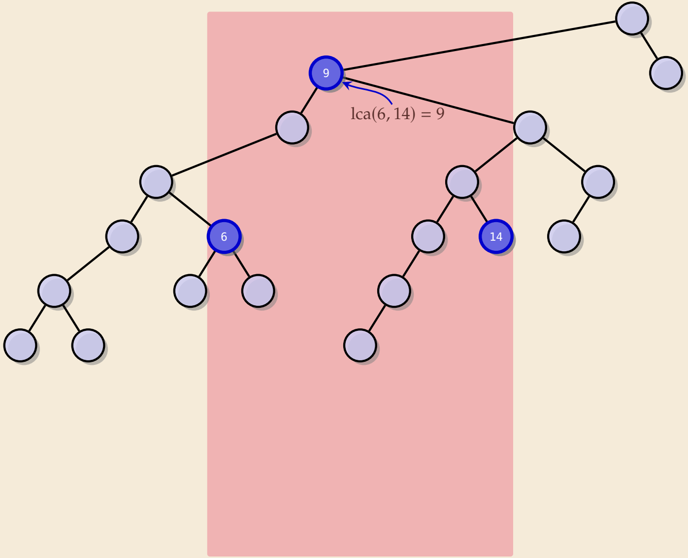
7.3 RMQ – Cartesian Trees

RMQ & LCA



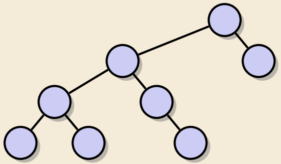
- **R**ange-**m**ax **q**ueries on array A:
  $$\mathrm{rmq}_A(i,j) = \arg\max_{i \le k \le j} A[k]$$
  $$= \textit{index} \text{ of max}$$
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!

RMQ & LCA



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

Counting binary trees



- Given the Cartesian tree, all RMQ answers are determined

  and vice versa!

- How many different Cartesian trees are there for arrays of length n?
  - known result: *Catalan numbers* $\dfrac{1}{n+1}\dbinom{2n}{n}$
  - easy to see: $\le 2^{2n}$

$\leadsto$ many arrays will give rise to the same Cartesian tree

Can we exploit that?

Intrablock queries

⇝ It remains to solve the intrablock queries!

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

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

- Choose $b = \left\lceil \frac{1}{4} \lg n \right\rceil$

- many blocks, but just b numbers long
  - ⇝ Cartesian tree of b elements can be encoded using $2b = \frac{1}{2} \lg n$ bits
  - ⇝ # different Cartesian trees is $\le 2^{2b} = 2^{\frac{1}{2} \lg n} = \left(2^{\lg n}\right)^{1/2} = \sqrt{n}$
  - ⇝ many *equivalent* blocks!

⇝ **Recall:** ***Exhaustive-Tabulation Technique:***

1. represent each subproblem by storing its *type* (here: encoding of Cartesian tree)
2. *enumerate* all possible subproblem types and their solutions
3. use type as index in a large *lookup table*

Exhaustive Tabulation

1. For each block, compute $2b$ bit representation of Cartesian tree
   - can be done in linear time
2. Compute large lookup table

Block type	i	j	$\mathrm{RMQ}(i,j)$
$\vdots$			
$\vdots$			

- $\le \sqrt{n}$ block types
- $\le b^2$ combinations for i and j

$\rightsquigarrow$ $\Theta\left(\sqrt{n} \cdot \log^2 n\right)$ rows

- each row can be computed in $O(\log n)$ time

$\rightsquigarrow$ overall preprocessing: $O(n)$ time!

RMQ 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

7.4 String Matching in Enhanced Suffix Array

Binary searching the suffix array

Recall: Can solve the string matching problem by binary searching $P[0..m)$ in $L[0..n]$

- worst-case cost: $O(\log(n) \cdot m)$ character comparisons

$\rightsquigarrow$ use LCP information to speed up string comparisons

- with RMQ on LCP array can determine lcp with middle

Conclusion

- *(Enhanced) Suffix Arrays* are the modern version of suffix trees
  - directly simulate suffix tree operations on L and LCP arrays

- can be harder to reason about
- can support same algorithms as suffix trees
- but use much less space
- simple(r) linear-time construction

Outlook:

- enhanced suffix arrays still need original text T to work
- a *self-index* avoids that
  - can store T in *compressed* form **and** support operations like string matching

7.5 The Burrows-Wheeler Transform

Digression: Recall BWT

Burrows-Wheeler Transform

1. Take all cyclic shifts of S
2. Sort cyclic shifts
3. Extract last column

S = `alf␣eats␣alfalfa$`

B = `asff$f␣e␣lllaaata`

```
alf␣eats␣alfalfa$
lf␣eats␣alfalfa$a
f␣eats␣alfalfa$al
␣eats␣alfalfa$alf
eats␣alfalfa$alf␣
ats␣alfalfa$alf␣e
ts␣alfalfa$alf␣ea
s␣alfalfa$alf␣eat
␣alfalfa$alf␣eats
alfalfa$alf␣eats␣
lfalfa$alf␣eats␣a
falfa$alf␣eats␣al
alfa$alf␣eats␣alf
lfa$alf␣eats␣alfa
fa$alf␣eats␣alfal
a$alf␣eats␣alfalf
$alf␣eats␣alfalfa
```

sort ⇝

BWT ↓

```
$alf␣eats␣alfalfa
␣alfalfa$alf␣eats
␣eats␣alfalfa$alf
a$alf␣eats␣alfalf
alf␣eats␣alfalfa$
alfa$alf␣eats␣alf
alfalfa$alf␣eats␣
ats␣alfalfa$alf␣e
eats␣alfalfa$alf␣
f␣eats␣alfalfa$al
fa$alf␣eats␣alfal
falfa$alf␣eats␣al
lf␣eats␣alfalfa$a
lfa$alf␣eats␣alfa
lfalfa$alf␣eats␣a
s␣alfalfa$alf␣eat
ts␣alfalfa$alf␣ea
```

Digression: Computing the BWT

How can we compute the BWT of a text efficiently?

- cyclic shifts S $\widehat{=}$ suffixes of S
  - comparing cyclic shifts stops at first $
  - for comparisons, anything after $ irrelevant!
- BWT is essentially suffix sorting!
  - $B[i] = S[L[i] - 1]$
  - where $L[i] = 0$, $B[i] =$ $

⇝ Can compute B in $O(n)$ time from L

	r	↓	$L[r]$
`alf␣eats␣alfalfa$`	0	`$alf␣eats␣alfalfa`	16
`lf␣eats␣alfalfa$a`	1	`␣alfalfa$alf␣eats`	8
`f␣eats␣alfalfa$al`	2	`␣eats␣alfalfa$alf`	3
`␣eats␣alfalfa$alf`	3	`a$alf␣eats␣alfalf`	15
`eats␣alfalfa$alf␣`	4	`alf␣eats␣alfalfa$`	0
`ats␣alfalfa$alf␣e`	5	`alfa$alf␣eats␣alf`	12
`ts␣alfalfa$alf␣ea`	6	`alfalfa$alf␣eats␣`	9
`s␣alfalfa$alf␣eat`	7	`ats␣alfalfa$alf␣e`	5
`␣alfalfa$alf␣eats`	8	`eats␣alfalfa$alf␣`	4
`alfalfa$alf␣eats␣`	9	`f␣eats␣alfalfa$al`	2
`lfalfa$alf␣eats␣a`	10	`fa$alf␣eats␣alfal`	14
`falfa$alf␣eats␣al`	11	`falfa$alf␣eats␣al`	11
`alfa$alf␣eats␣alf`	12	`lf␣eats␣alfalfa$a`	1
`lfa$alf␣eats␣alfa`	13	`lfa$alf␣eats␣alfa`	13
`fa$alf␣eats␣alfal`	14	`lfalfa$alf␣eats␣a`	10
`a$alf␣eats␣alfalf`	15	`s␣alfalfa$alf␣eat`	7
`$alf␣eats␣alfalfa`	16	`ts␣alfalfa$alf␣ea`	6

7.6 Inverting the BWT

7.7 Searching in the BWT