ALGORITHMS OF BIOINFORMATICS

7

Googling Genomes

15 January 2026

Prof. Dr. Sebastian Wild

CS594 (Winter 2025-26)
Philipps-Universität Marburg
version 2026-01-22 14:03

Outline

7 Googling Genomes

Application 4: Longest Common Extensions

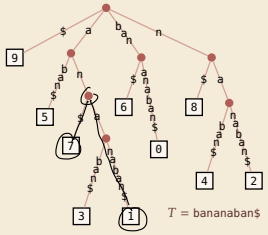
- We implicitly used a special case of a more general, versatile idea:

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



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

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 ...

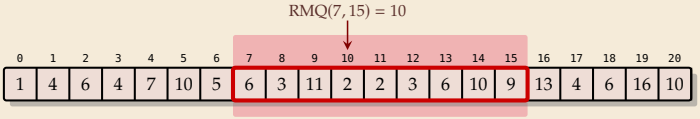
$\rightsquigarrow$ for now, use $O(1)$ LCA as black box.

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

7.1 Range-Minimum Queries

Range-minimum queries (RMQ)

- **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

Finally: Longest common extensions

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

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

Inverse suffix array: going left & right

- to understand the fastest algorithm, it is helpful to define the *inverse suffix array*:
  - $R[i] = r \iff L[r] = i$ $\quad L$ = *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	$

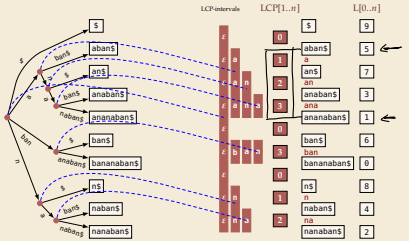
right: $R[0] = 6$ — left: $L[8] = 4$

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$

sort suffixes

29

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 $\leq P(n)$)
  2. **Query time**: Running time $Q(n)$ of one query (using precomputed data)

- Write $\langle P(n), Q(n) \rangle$ **time solution** for short

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 $\leq P(n)$)
  2. **Query time**: Running time $Q(n)$ of one query (using precomputed data)

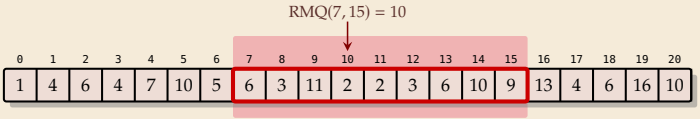
- Write $\langle P(n), Q(n) \rangle$ **time solution** for short

RMQ Implications for LCE

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

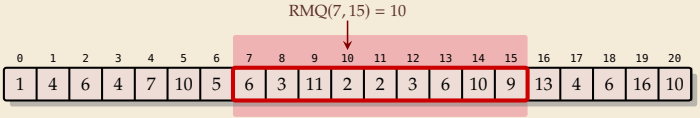
$\rightsquigarrow$ $\langle P(n), Q(n) \rangle$ time RMQ data structure implies
$\langle P(n) + O(n), Q(n) \rangle$ time LCE data structure

Trivial Solutions



- Two easy solutions show extreme ends of scale:

Trivial Solutions

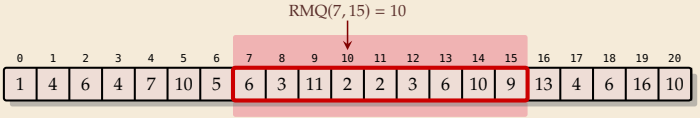


- Two easy solutions show extreme ends of scale:

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

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

Trivial Solutions



- Two easy solutions show extreme ends of scale:

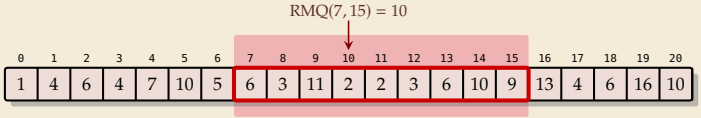
1. **Scan on demand**
   - no preprocessing at all
   - answer $\mathrm{RMQ}(i, j)$ by scanning through $A[i..j]$, keeping track of min

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

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

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

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.
  - $\rightsquigarrow \leq n \cdot \lg n$ entries
  - $\rightsquigarrow$ Can be stored as $M'[i][k]$

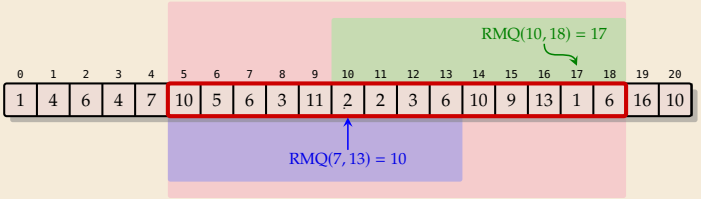
Sparse Table

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

Sparse Table

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

- How to answer queries?

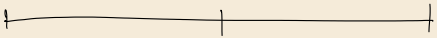


1. Find k with $\ell/2 \leq 2^k \leq \ell$
2. Cover range $[i..j]$ by
   2^k positions right from i and
   2^k positions left from j
3. $\mathrm{RMQ}(i, j) =$
   $\arg\min\{A[rmq_1], A[rmq_2]\}$

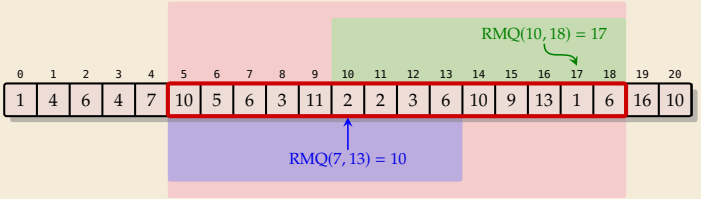
   with $rmq_1 = \mathrm{RMQ}(i, i + 2^k - 1)$
   $rmq_2 = \mathrm{RMQ}(j - 2^k + 1, j)$

Sparse Table

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



- 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!

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

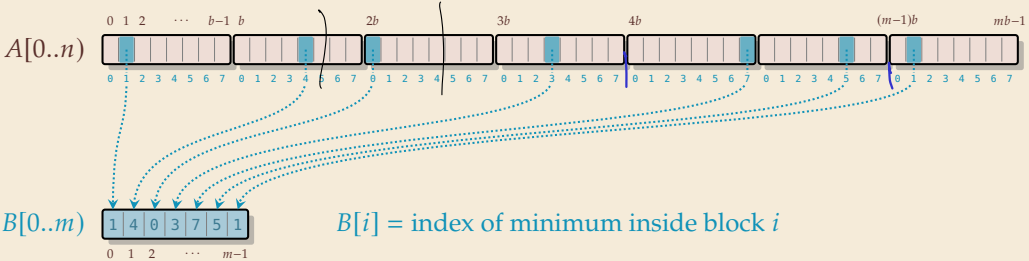
Bootstrapping

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

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, $O(m \log m) = O(n)$!

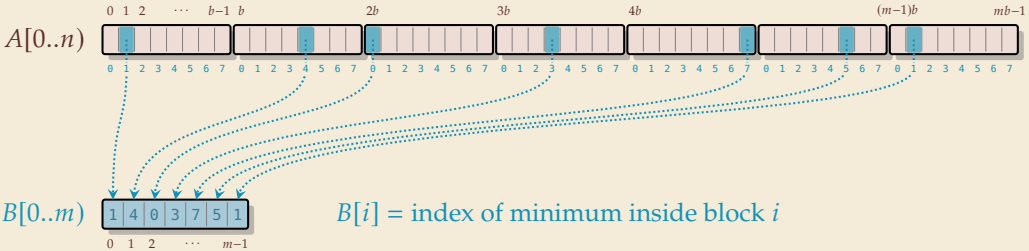
- Break A into blocks of $b = O(\log n)$ numbers
- Create array of block minima $B[0..m)$ for $m = \lceil n/b \rceil = O(n/\log n)$



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 $m = \lceil n/b \rceil = O(n/\log n)$

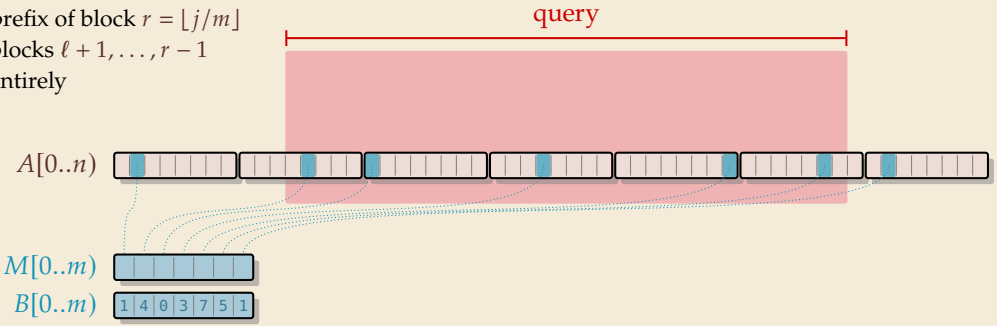


  - ⇝ Use sparse table solution for B.

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

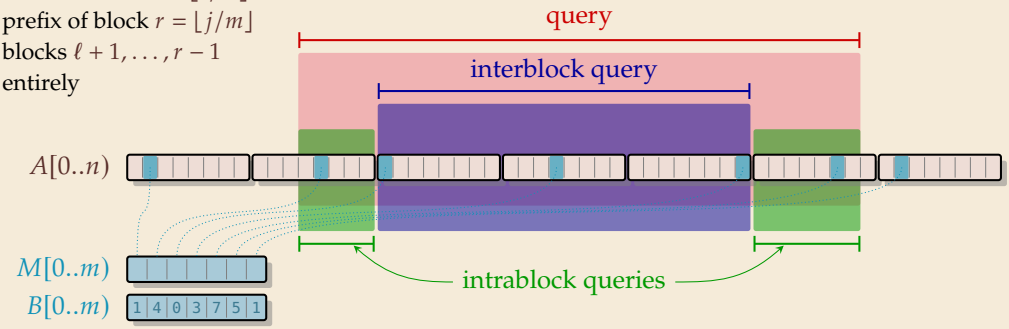
Query decomposition

- 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



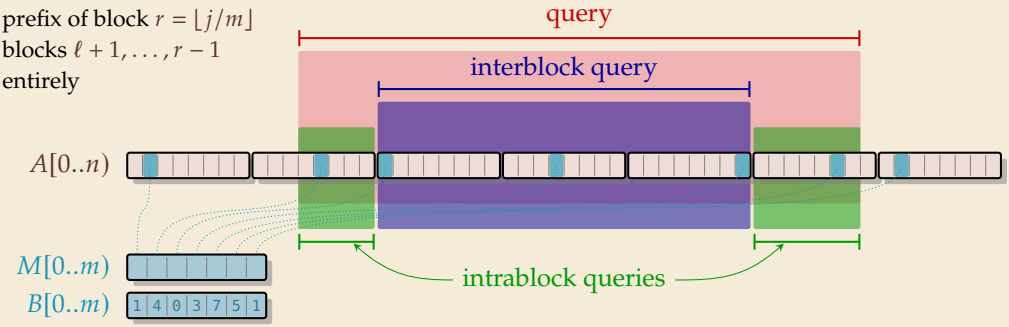
Query decomposition

- 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



Query decomposition

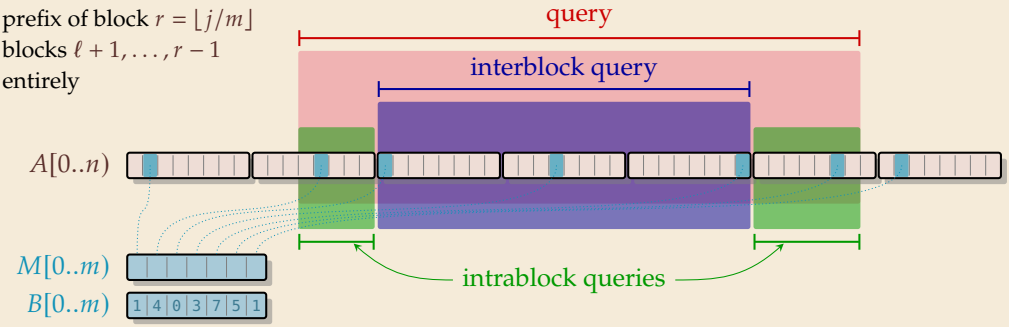
- Query $\mathrm{RMQ}_A(i, j)$ covers
  - suffix of block $\ell = \lfloor i/m \rfloor$
  - prefix of block $r = \lfloor j/m \rfloor$
  - blocks $\ell + 1, \ldots, r - 1$ entirely



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

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

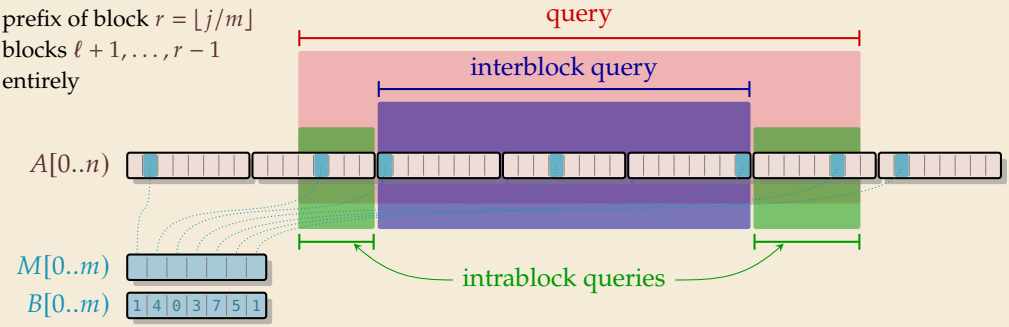


- $\mathrm{RMQ}_A(i, j) = \arg\min_{k \in K} A[k]$ with $K = \begin{Bmatrix} \mathrm{RMQ}_{\text{block } \ell}(i - \ell b,\ (\ell + 1)b - 1), \\ b \cdot \mathrm{RMQ}_M(\ell + 1, r - 1) + \\ \quad B\left[\mathrm{RMQ}_M(\ell + 1, r - 1)\right], \\ \mathrm{RMQ}_{\text{block } r}(rb,\ j - rb) \end{Bmatrix}$

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

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



- $\mathrm{RMQ}_A(i, j) = \arg\min_{k \in K} A[k]$ with $K = \begin{Bmatrix} \mathrm{RMQ}_{\mathrm{block}\,\ell}(i - \ell b,\ (\ell + 1)b - 1), \\ b \cdot \mathrm{RMQ}_M(\ell + 1, r - 1) + \\ \quad B\left[\mathrm{RMQ}_M(\ell + 1, r - 1)\right], \\ \mathrm{RMQ}_{\mathrm{block}\,r}(rb,\ j - rb) \end{Bmatrix}$

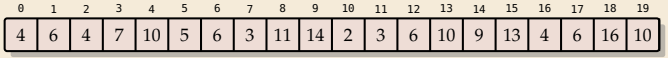
⇝ only 3 possible values to check if intrablock and interblock queries known

$O(\log n)$

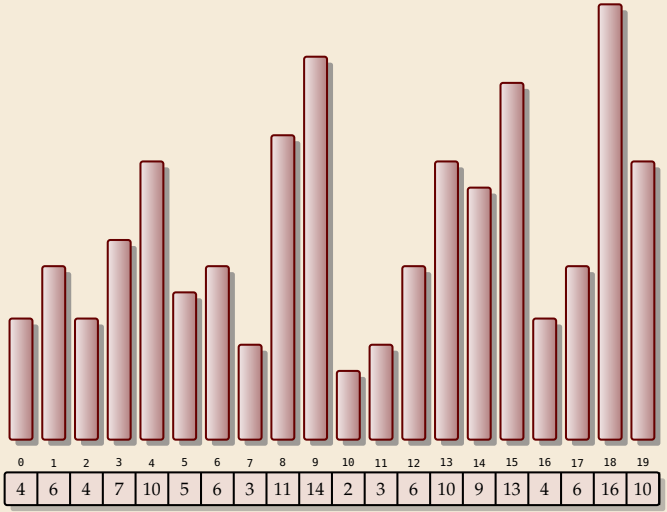
$\langle O(n),\ O(\log n) \rangle$

7.3 RMQ – Cartesian Trees

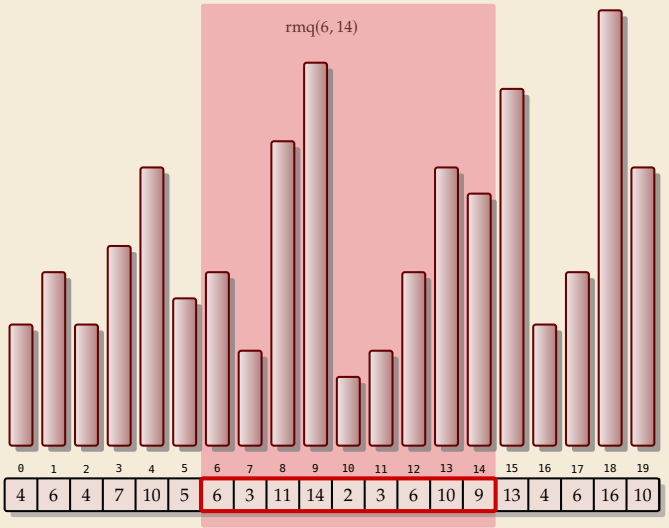
RMQ & LCA


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

RMQ & LCA

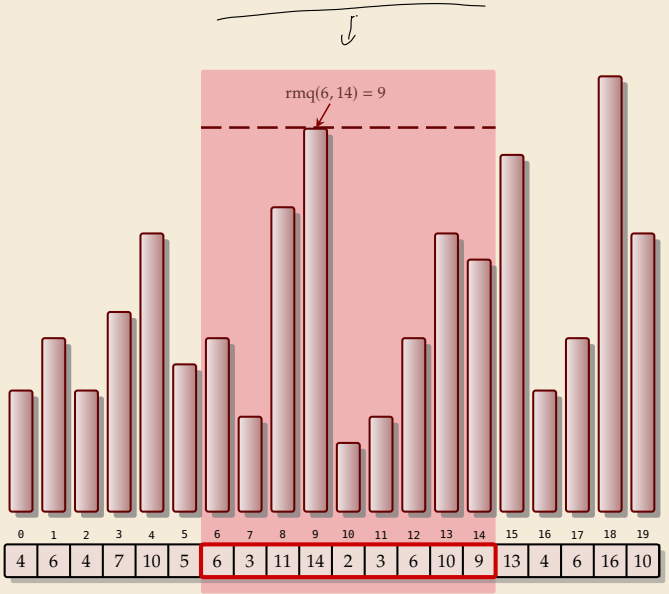

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
4 6 4 7 10 5 6 3 11 14 2 3 6 10 9 13 4 6 16 10

RMQ & LCA



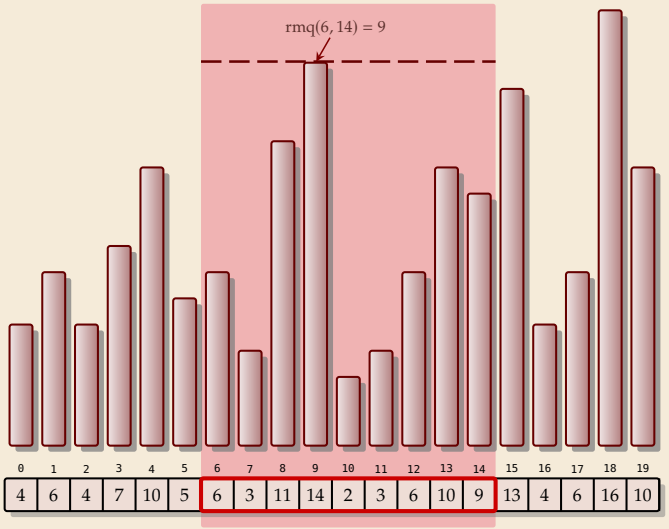
- **Range-max queries** on array A:
  $\mathrm{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  $=$ *index* of max

RMQ & LCA



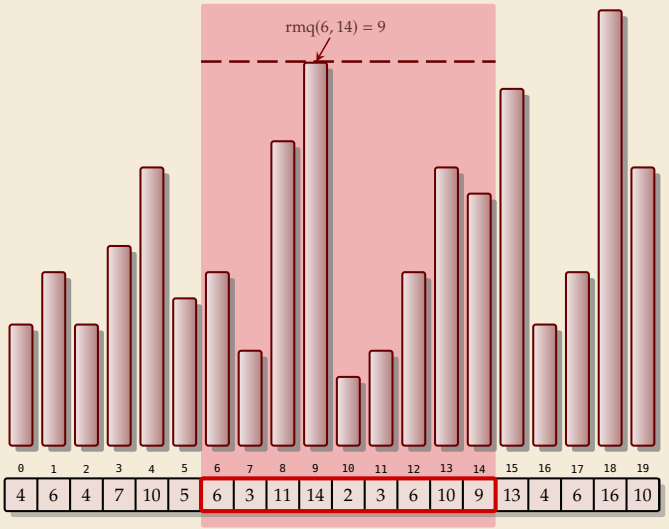
- **Range-max queries** on array A:
  $\mathrm{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  = *index* of max

RMQ & LCA



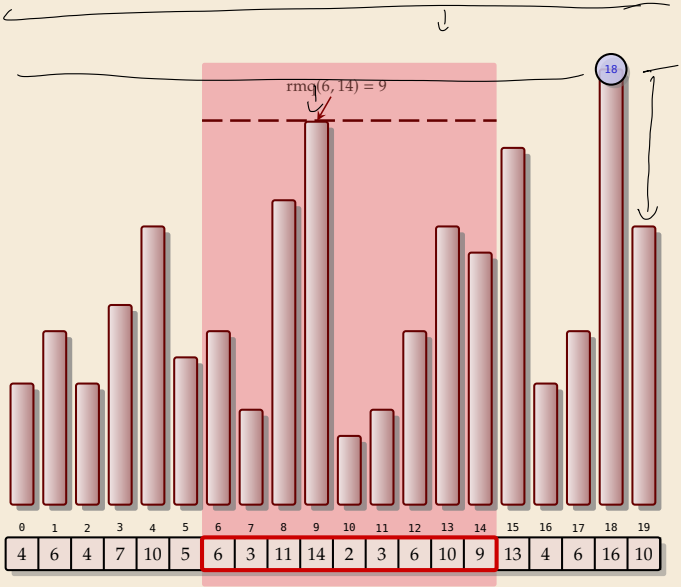
- **Range-max queries** on array A:
  $\mathrm{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  $=$ *index* of max
- **Task:** Preprocess A,
  then answer RMQs fast

RMQ & LCA



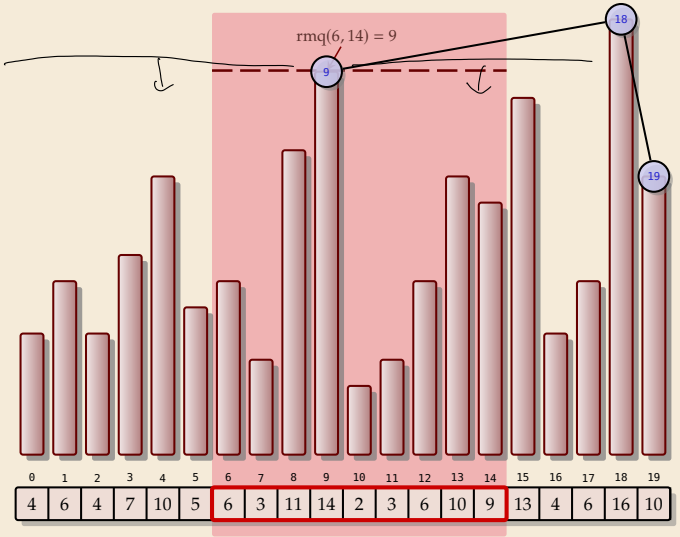
- **Range-max queries** on array A:
  $\mathrm{rmq}_A(i,j) = \arg\max_{i \le k \le j} A[k]$
  $=$ *index* of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!

RMQ & LCA



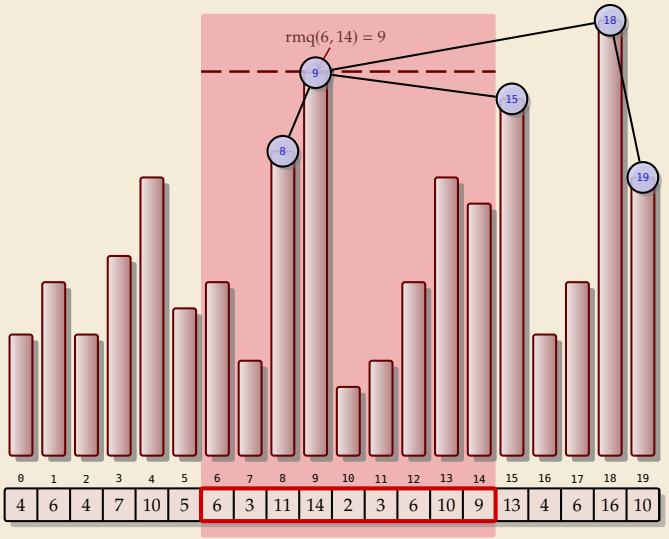
- **Range-max queries** on array A:
  $\mathrm{rmq}_A(i,j) = \arg\max_{i \le k \le j} A[k]$
  $=$ *index* of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!
- **Cartesian tree:** (cf. *treap*)
  construct binary tree by
  sweeping line down

RMQ & LCA



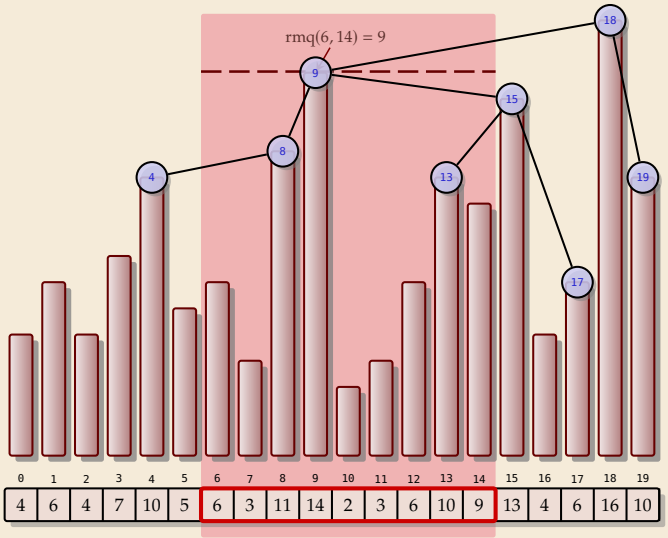
- **Range-max queries** on array A:
  $\text{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  $=$ *index* of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!
- **Cartesian tree:** (cf. *treap*)
  construct binary tree by
  sweeping line down

RMQ & LCA



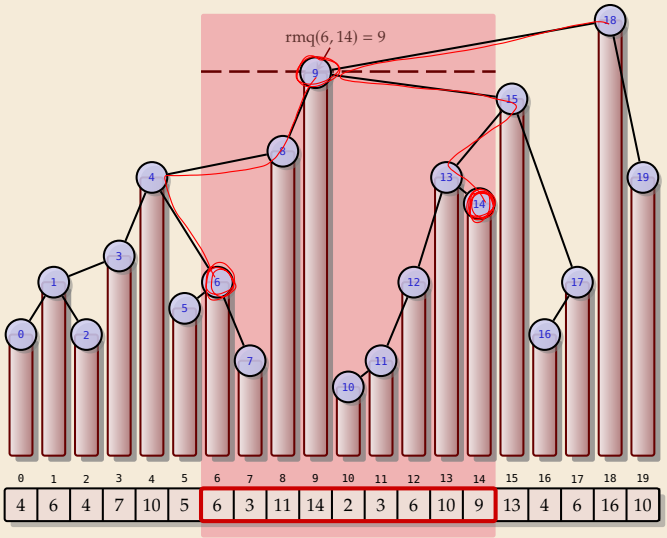
- **Range-max queries** on array A:
  $\mathrm{rmq}_A(i,j) = \arg\max_{i \le k \le j} A[k]$
  $=$ *index* of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!
- **Cartesian tree:** (cf. *treap*)
  construct binary tree by
  sweeping line down

RMQ & LCA



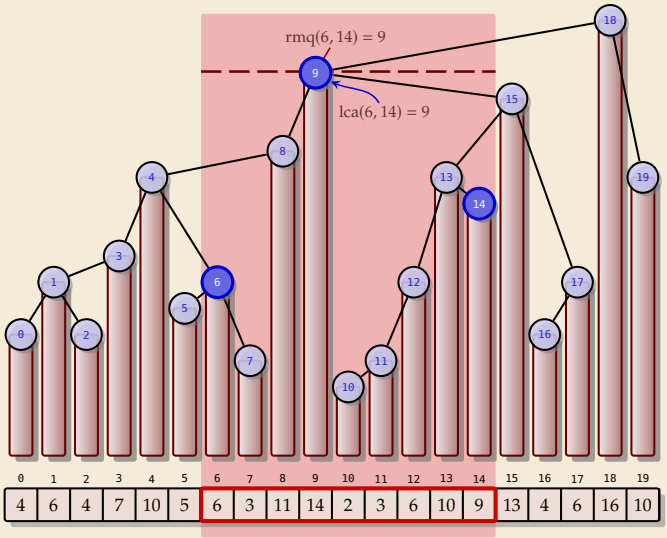
- **Range-max queries** on array A:
  $\mathrm{rmq}_A(i,j) = \arg\max_{i \le k \le j} A[k]$
  $=$ *index* of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!
- **Cartesian tree:** (cf. *treap*)
  construct binary tree by
  sweeping line down

RMQ & LCA



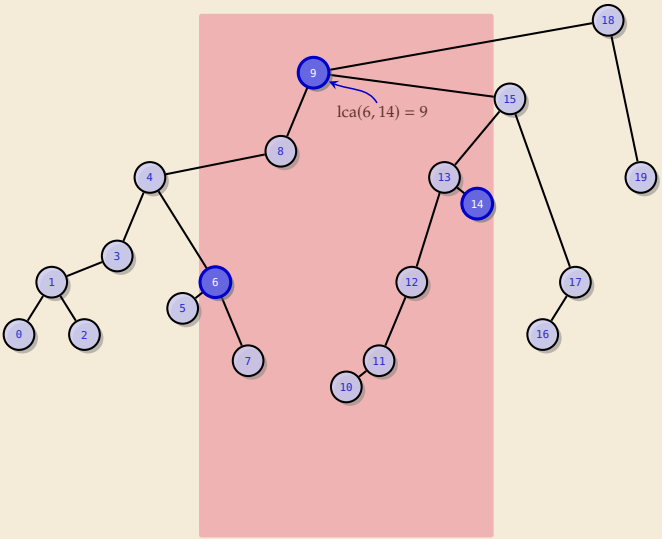
- **Range-max queries** on array A:
  $\mathrm{rmq}_A(i, j) = \arg\max_{i \le k \le j} A[k]$
  $= \textit{index}$ of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!
- **Cartesian tree:** (cf. *treap*)
  construct binary tree by
  sweeping line down

RMQ & LCA



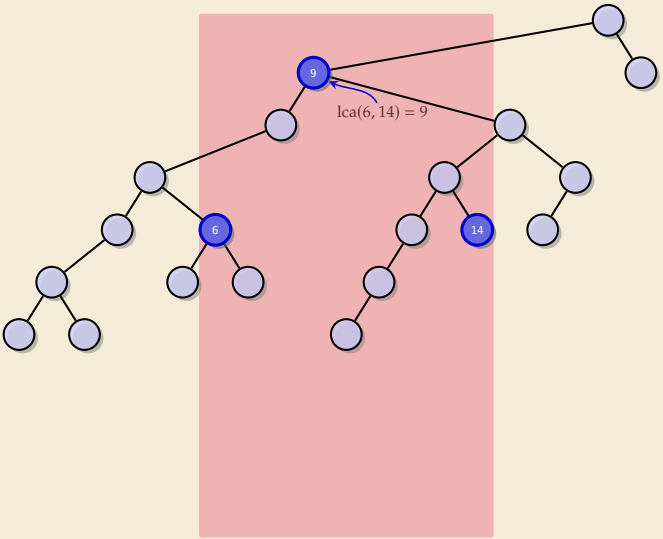
- **Range-max queries** on array A:
  $\text{rmq}_A(i,j) = \arg\max_{i \le k \le j} A[k]$
  $=$ *index* of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!
- **Cartesian tree:** (cf. *treap*)
  construct binary tree by
  sweeping line down
- $\text{rmq}(i,j)$ =
  lowest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

RMQ & LCA



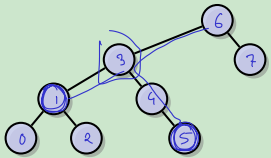
- **Range-max queries** on array A:
  $\text{rmq}_A(i,j) = \arg\max_{i \le k \le j} A[k]$
  $=$ *index* of max
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!
- **Cartesian tree:** (cf. *treap*)
  construct binary tree by
  sweeping line down
- $\text{rmq}(i,j)$ $=$
  lowest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

RMQ & LCA



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

Clicker Question

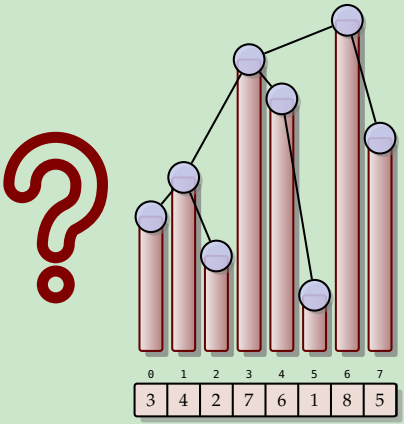


① Inorder traversal

Given the (max-oriented) Cartesian tree for A on the left, what is $\mathrm{RMQ}_A(1,5)$?

→ sli.do/cs594

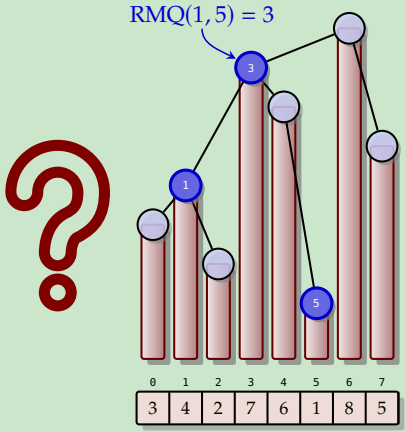
Clicker Question



Given the (max-oriented) Cartesian tree for A on the left, what is $\mathrm{RMQ}_A(1,5)$?

→ *sli.do/cs594*

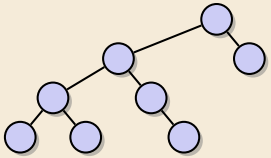
Clicker Question



Given the (max-oriented) Cartesian tree for A on the left, what is $\mathrm{RMQ}_A(1,5)$?

→ *sli.do/cs594*

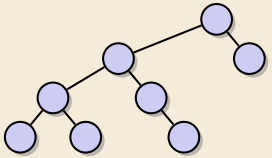
Counting binary trees



- Given the Cartesian tree,
  all RMQ answers are determined

  and vice versa!

Counting binary trees



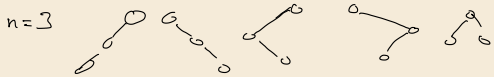
- Given the Cartesian tree,
  all RMQ answers are determined

  and vice versa!

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

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

Can we exploit that?



$$\frac{1}{4}\cdot\binom{6}{3} = \frac{6!}{3!\,3!\,4} = \frac{6\cdot 5}{3!} = 5$$

Encoding binary trees: in preorder traversal write down (has left, has right)

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!

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 $\leq 2^{2b} = 2^{\frac{1}{2} \lg n} = \left(2^{\lg n}\right)^{1/2} = \sqrt{n}$
  - ⇝ many *equivalent* blocks!

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 $\leq 2^{2b} = 2^{\frac{1}{2} \lg n} = \left(2^{\lg n}\right)^{1/2} = \sqrt{n}$
  - ⇝ 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

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	RMQ(i, j)
⋮			
 01 10 00	0	1	0
	0	2	0
	1	2	2
⋮			

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	RMQ(i, j)
⋮			
⋮			

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

$\rightsquigarrow \Theta(\sqrt{n} \cdot \log^2 n)$ 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!

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

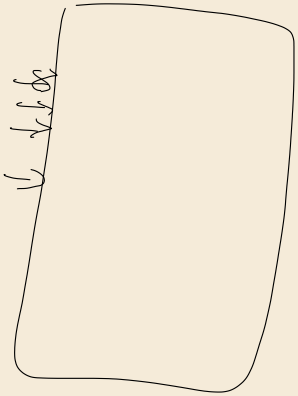
7.4 String Matching in Enhanced Suffix Array

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: $\langle \lg n + 2 \rangle$ *string* comparisons of string of length m

$\rightsquigarrow$ $O(\log(n) \cdot m)$ character comparisons



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: $\lg n + 2$ *string* comparisons of string of length m

$\rightsquigarrow$ $O(\log(n) \cdot m)$ character comparisons

- suffix tree could do $O(m)$ total time (assuming constant σ or hashing for child links)
- surely, enhanced suffix arrays can do better than $O(m \log n)$ 🤔

Binary searching the suffix array

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

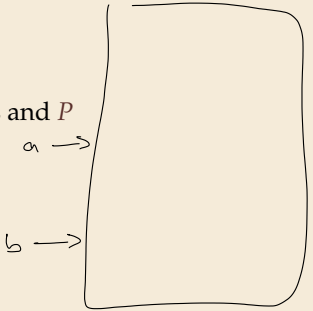
- worst-case cost: $\lg n + 2$ *string* comparisons of string of length m

$\rightsquigarrow$ $O(\log(n) \cdot m)$ character comparisons

- suffix tree could do $O(m)$ total time (assuming constant σ or hashing for child links)
- surely, enhanced suffix arrays can do better than $O(m \log n)$ 😉

Idea: use LCP information to save character comparisons

- concretely: maintain LCP between lower/upper bound suffixes and P
  $T[a..n] \leq_{\text{lex}} P \leq_{\text{lex}} T[b..n]$
  $\ell_a = \text{LCP}(T[a..n], P)$ and $\ell_b = \text{LCP}(T[b..n], P)$



Binary searching the suffix array

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

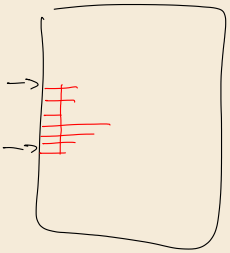
- worst-case cost: $\lg n + 2$ *string* comparisons of string of length m

$\rightsquigarrow$ $O(\log(n) \cdot m)$ character comparisons

- suffix tree could do $O(m)$ total time (assuming constant σ or hashing for child links)
- surely, enhanced suffix arrays can do better than $O(m \log n)$ 😉

Idea: use LCP information to save character comparisons

- concretely: maintain LCP between lower/upper bound suffixes and P
  $T[a..n] \leq_{\text{lex}} P \leq_{\text{lex}} T[b..n]$
  $\ell_a = \text{LCP}(T[a..n], P)$ and $\ell_b = \text{LCP}(T[b..n], P)$



- avoid comparing same characters again
- Note: with RMQ on LCP array can determine $\text{LCP}(T_i, T_j)$ for any $i, j \in [0..n)$

LCP Binary Search

- Input: $\ell_a = \mathrm{LCP}(T_a, P)$
  $\ell_b = \mathrm{LCP}(T_b, P)$

  $\rightsquigarrow\ \ell_m = \mathrm{LCP}(T_m, P) \geq \min\{\ell_a, \ell_b\}$

0:	T_{20}	`$`
1:	T_4	`ahbansbananasman$`
a → 2:	T_{18}	`an$`
3:	T_{11}	`ananasman$`
4:	T_{13}	`anasman$`
5:	T_1	`annahbansbananasman$`
6:	T_7	`ansbananasman$`
r → 7:	T_{15}	`asman$`
8:	T_{10}	`bananasman$`
9:	T_6	`bansbananasman$`
10:	T_0	`hannahbansbananasman$`
11:	T_5	`hbansbananasman$`
12:	T_{17}	`man$`
b → 13:	T_{19}	`n$`
14:	T_3	`nahbansbananasman$`
15:	T_{12}	`nanasman$`
16:	T_{14}	`nasman$`
17:	T_2	`nnahbansbananasman$`
18:	T_8	`nsbananasman$`
19:	T_9	`sbananasman$`
20:	T_{16}	`sman$`

LCP Binary Search

- Input: $\ell_a = \mathrm{LCP}(T_a, P)$
  $\ell_b = \mathrm{LCP}(T_b, P)$

  $\rightsquigarrow\ \ell_m = \mathrm{LCP}(T_m, P) \geq \min\{\ell_a, \ell_b\}$

- **Case 1:** $\ell_a = \ell_b$
  Compare P and T_m starting at ℓ_a

0:	T_{20}		$	
1:	T_4	$a \rightarrow$	ahbansbananasman$	**Case 1**
2:	T_{18}		an$	
3:	T_{11}		ananasman$	
4:	T_{13}	$m \rightarrow$	anasman$	P = anna
5:	T_1	⇓	annahbansbananasman$	
6:	T_7		ansbananasman$	
7:	T_{15}	$b \rightarrow$	asman$	
8:	T_{10}		bananasman$	
9:	T_6		bansbananasman$	
10:	T_0		hannahbansbananasman$	
11:	T_5		hbansbananasman$	
12:	T_{17}		man$	
13:	T_{19}		n$	
14:	T_3		nahbansbananasman$	
15:	T_{12}		nanasman$	
16:	T_{14}		nasman$	
17:	T_2		nnahbansbananasman$	
18:	T_8		nsbananasman$	
19:	T_9		sbananasman$	
20:	T_{16}		sman$	

LCP Binary Search

- Input: $\ell_a = \mathrm{LCP}(T_a, P)$
  $\ell_b = \mathrm{LCP}(T_b, P)$

  $\rightsquigarrow\ \ell_m = \mathrm{LCP}(T_m, P) \geq \min\{\ell_a, \ell_b\}$

- **Case 1:** $\ell_a = \ell_b$
  Compare P and T_m starting at ℓ_a

- **Case 2:** $\ell_a \neq \ell_b$; w.l.o.g. $\ell_a > \ell_b$
  - **Case 2a:** $\mathrm{LCP}(T_a, T_m) > \ell_a$
    $P >_{\mathrm{lex}} T_m$ w/o cmps!

0:	T_{20}		$	
1:	T_4		ahbansbananasman$	
2:	T_{18}		an$	
3:	T_{11}		ananasman$	
4:	T_{13}	$a \rightarrow$	anasman$	Case 2a
5:	T_1		annahbansbananasman$	
6:	T_7	$m \rightarrow$	ansbananasman$	P = asterix
7:	T_{15}		asman$	
8:	T_{10}	$b \rightarrow$	bananasman$	
9:	T_6		bansbananasman$	
10:	T_0		hannahbansbananasman$	
11:	T_5		hbansbananasman$	
12:	T_{17}		man$	
13:	T_{19}		n$	
14:	T_3		nahbansbananasman$	
15:	T_{12}		nanasman$	
16:	T_{14}		nasman$	
17:	T_2		nnahbansbananasman$	
18:	T_8		nsbananasman$	
19:	T_9		sbananasman$	
20:	T_{16}		sman$	

LCP Binary Search

- Input: $\ell_a = \mathrm{LCP}(T_a, P)$
  $\ell_b = \mathrm{LCP}(T_b, P)$

  $\rightsquigarrow\ \ell_m = \mathrm{LCP}(T_m, P) \geq \min\{\ell_a, \ell_b\}$

- **Case 1:** $\ell_a = \ell_b$
  Compare P and T_m starting at ℓ_a
- **Case 2:** $\ell_a \neq \ell_b$; w.l.o.g. $\boxed{\ell_a > \ell_b}$
  - **Case 2a:** $\mathrm{LCP}(T_a, T_m) > \ell_a$
    $P >_{\mathrm{lex}} T_m$ w/o cmps!
  - **Case 2b:** $\mathrm{LCP}(T_a, T_m) < \ell_a$
    $P <_{\mathrm{lex}} T_m$ w/o cmps!

0:	T_{20}		$	
1:	T_4		ahbansbananasman$	
2:	T_{18}		an$	
3:	T_{11}	$a \longrightarrow$	ananasman$	Case 2b
4:	T_{13}	$m \longrightarrow$	anasman$	P = ananasmen
5:	T_1		annahbansbananasman$	
6:	T_7	$b \longrightarrow$	ansbananasman$	
7:	T_{15}		asman$	
8:	T_{10}		bananasman$	
9:	T_6		bansbananasman$	
10:	T_0		hannahbansbananasman$	
11:	T_5		hbansbananasman$	
12:	T_{17}		man$	
13:	T_{19}		n$	
14:	T_3		nahbansbananasman$	
15:	T_{12}		nanasman$	
16:	T_{14}		nasman$	
17:	T_2		nnahbansbananasman$	
18:	T_8		nsbananasman$	
19:	T_9		sbananasman$	
20:	T_{16}		sman$	

LCP Binary Search

- Input: $\ell_a = \mathrm{LCP}(T_a, P)$
  $\ell_b = \mathrm{LCP}(T_b, P)$

  $\rightsquigarrow\ \ell_m = \mathrm{LCP}(T_m, P) \geq \min\{\ell_a, \ell_b\}$

- **Case 1:** $\ell_a = \ell_b$
  Compare P and T_m starting at ℓ_a

- **Case 2:** $\ell_a \neq \ell_b$; w.l.o.g. $\boxed{\ell_a > \ell_b}$
  - **Case 2a:** $\mathrm{LCP}(T_a, T_m) > \ell_a$
    $P >_{\mathrm{lex}} T_m$ w/o cmps!
  - **Case 2b:** $\mathrm{LCP}(T_a, T_m) < \ell_a$
    $P <_{\mathrm{lex}} T_m$ w/o cmps!
  - **Case 2c:** $\mathrm{LCP}(T_a, T_m) = \ell_a$
    Compare P and T_m from ℓ_a

0:	T_{20}		$	
1:	T_4		ahbansbananasman$	
2:	T_{18}		an$	
3:	T_{11}	$a \rightarrow$	ananasman$	Case 2c
4:	T_{13}	$m \rightarrow$	anasman$	P = anarchy
5:	T_1		annahbansbananasman$	
6:	T_7	$b \rightarrow$	ansbananasman$	
7:	T_{15}		asman$	
8:	T_{10}		bananasman$	
9:	T_6		bansbananasman$	
10:	T_0		hannahbansbananasman$	
11:	T_5		hbansbananasman$	
12:	T_{17}		man$	
13:	T_{19}		n$	
14:	T_3		nahbansbananasman$	
15:	T_{12}		nanasman$	
16:	T_{14}		nasman$	
17:	T_2		nnahbansbananasman$	
18:	T_8		nsbananasman$	
19:	T_9		sbananasman$	
20:	T_{16}		sman$	

LCP Binary Search

- Input: $\ell_a = \mathrm{LCP}(T_a, P)$
  $\ell_b = \mathrm{LCP}(T_b, P)$

  $\rightsquigarrow\ \ell_m = \mathrm{LCP}(T_m, P) \geq \min\{\ell_a, \ell_b\}$

- **Case 1:** $\ell_a = \ell_b$
  Compare P and T_m starting at ℓ_a

- **Case 2:** $\ell_a \neq \ell_b$; w.l.o.g. $\boxed{\ell_a > \ell_b}$
  - **Case 2a:** $\mathrm{LCP}(T_a, T_m) > \ell_a$
    $P >_{\mathrm{lex}} T_m$ w/o cmps!
  - **Case 2b:** $\mathrm{LCP}(T_a, T_m) < \ell_a$
    $P <_{\mathrm{lex}} T_m$ w/o cmps!
  - **Case 2c:** $\mathrm{LCP}(T_a, T_m) = \ell_a$
    Compare P and T_m from ℓ_a

- in each case, learn ℓ_m $\rightsquigarrow$ invariant

- no redundant '='-comparisons

0:	T_{20}		$	
1:	T_4		ahbansbananasman$	
2:	T_{18}		an$	
3:	T_{11}	$a \longrightarrow$	ananasman$	Case 2c
4:	T_{13}	$m \longrightarrow$	anasman$	P = anarchy
5:	T_1		annahbansbananasman$	
6:	T_7	$b \longrightarrow$	ansbananasman$	
7:	T_{15}		asman$	
8:	T_{10}		bananasman$	
9:	T_6		bansbananasman$	
10:	T_0		hannahbansbananasman$	
11:	T_5		hbansbananasman$	
12:	T_{17}		man$	
13:	T_{19}		n$	
14:	T_3		nahbansbananasman$	
15:	T_{12}		nanasman$	
16:	T_{14}		nasman$	
17:	T_2		nnahbansbananasman$	
18:	T_8		nsbananasman$	
19:	T_9		sbananasman$	
20:	T_{16}		sman$	

Enhanced Suffix Arrays – Update

- *Enhanced suffix array*: L, R and LCP array with RMQ support
- **Goal:** simulate any suffix tree operations
  - string matching in $O(m + \log n)$ time
  - string depth of internal nodes = LCP values
  - internal suffix tree node = LCP interval
    - ⇝ storing information per node
    - bottom-up traversal via enclosing LCP intervals
  - longest common extension queries
  - suffix links

Enhanced Suffix Arrays – Update

- *Enhanced suffix array*: L, R and LCP array with RMQ support
- **Goal:** simulate any suffix tree operations
  - string matching in $O(m + \log n)$ time ✓
  - string depth of internal nodes = LCP values ✓
  - internal suffix tree node = LCP interval ✓
    - ⇝ storing information per node ✓
    - bottom-up traversal via enclosing LCP intervals ✓
  - longest common extension queries ✓
  - suffix links ✓

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

Towards Self-Indexes

- For large genomes or multiple-genome datasets, can't hold $T[0..n)$ in fast memory.
- An enhanced suffix array needs additional $\Theta(n)$ words of space.

$\rightsquigarrow$ When reference genomes first became available, a major show stopper!

Towards Self-Indexes

- For large genomes or multiple-genome datasets, can't hold $T[0..n)$ in fast memory.
- An enhanced suffix array needs additional $\Theta(n)$ words of space.

$\rightsquigarrow$ When reference genomes first became available, a major show stopper!

- But since string matching can reconstruct T, can't avoid storing T somehow!
- A *self-index* is a data structure that answers operations without access to T at query time
  - We get to decide *how* to store T $\rightsquigarrow$ might *compress* T (if compressible)
  - Known as *"encoding model"* in space-efficient data structures

genomes highly repetitive!

Towards Self-Indexes

- For large genomes or multiple-genome datasets, can't hold $T[0..n)$ in fast memory.
- An enhanced suffix array needs additional $\Theta(n)$ words of space.

$\rightsquigarrow$ When reference genomes first became available, a major show stopper!

- But since string matching can reconstruct T, can't avoid storing T somehow!
- A *self-index* is a data structure that answers operations without access to T at query time
  - We get to decide *how* to store T $\rightsquigarrow$ might *compress* T (if compressible)
  - Known as *"encoding model"* in space-efficient data structures

genomes highly repetitive!

$\rightsquigarrow$ ***Key question:*** *How to compress T while supporting random access and read mapping?*
"Computing over compressed data"

BWT – Definitions

$T =$ `time␣flies␣quickly␣`

`flies␣quickly␣time␣`

- *cyclic shift* of a string:

⇝ cyclic shift

BWT – Definitions

- *cyclic shift* of a string:
- with end-of-word character `$`

⇝ can **recover** original string

T = `time␣flies␣quickly␣`

⇝ cyclic shift

`flies␣quickly␣time␣`

BWT – Definitions

- *cyclic shift* of a string:
- with end-of-word character `$`

⇝ can **recover** original string

T = `time␣flies␣quickly␣`

⇝ cyclic shift

`flies␣quickly␣time␣`

- The Burrows-Wheeler Transform proceeds in three steps:
  0. Append end-of-word character `$` to S.
  1. Consider *all cyclic shifts* of S
  2. Sort these strings lexicographically
  3. B is the *list of trailing characters* (last column, top-down) of each string

BWT – Example

S = `alf␣eats␣alfalfa$`

1. Take all cyclic shifts of S

```
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 – Example

S = `alf␣eats␣alfalfa$`

1. Take all cyclic shifts of S
2. Sort cyclic shifts

BWT – Example

S = `alf␣eats␣alfalfa$`

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

B = `asff$f␣e␣lllaaata`

Computing the BWT

How can we compute the BWT of a text efficiently?

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!

	r	$\downarrow L[r]$
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

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 $L[i] = 0$, $B[i] = \$$

```
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
```

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

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 $L[i] = 0$, $B[i] = \$$

$\rightsquigarrow$ Can compute B in $O(n)$ time from L

- more direct methods now also available

	r	$\downarrow L[r]$
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

BWT – Properties

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

Why does BWT help for compression?

- sorting *groups* characters *by what follows*

BWT – Properties

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

Why does BWT help for compression?

- sorting *groups* characters *by what follows*
  - Example: `lf` always preceded by `a`
  - more generally: BWT can be partitioned into letters following a given context

BWT – Properties

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

Why does BWT help for compression?

- sorting *groups* characters *by what follows*
  - Example: `lf` always preceded by `a`
  - more generally: BWT can be partitioned into letters following a given context

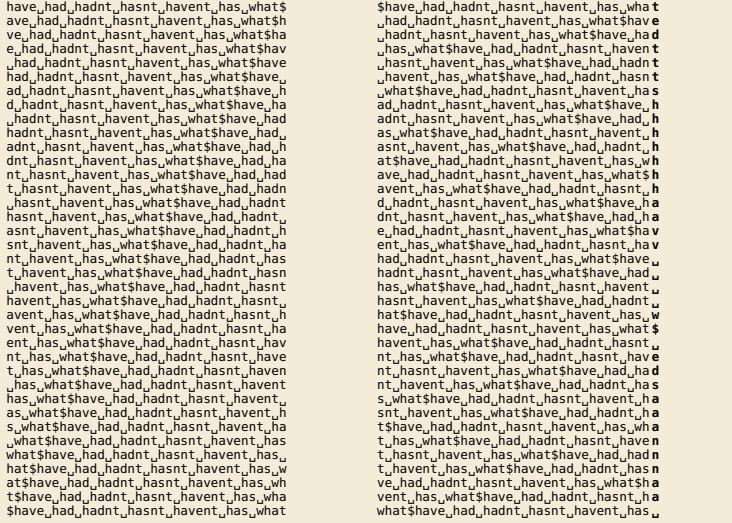
(formally: low higher-order empirical entropy)

⇝ *If S allows predicting symbols from context, B has locally low entropy of characters.*

- that makes MTF (move-to-front) transformation effective!

⇝ use in compression pipeline for bzip2: BTW → MTF → RLE → Huffman

A Bigger Example



$T =$ h a v e ␣ h a d ␣ h a d n t ␣ h a s n t ␣ h a v e n t ␣ h a s ␣ w h a t $

$B =$ t e d t t t s h h h h h h h a a v v ␣ ␣ ␣ ␣ w $ ␣ e d s a a a n n n a a ␣

Run-length BWT Compression

- amazingly, just run-length compressing the BWT is already powerful!
- r = number of runs in BWT

Run-length BWT Compression

- amazingly, just run-length compressing the BWT is already powerful!
- r = number of runs in BWT

Example:

S = `alf␣eats␣alfalfa$`

B = `asff$f␣e␣lllaaata`

$RL(B) = \begin{bmatrix}\texttt{a}\\1\end{bmatrix}\begin{bmatrix}\texttt{s}\\1\end{bmatrix}\begin{bmatrix}\texttt{f}\\2\end{bmatrix}\begin{bmatrix}\texttt{\$}\\1\end{bmatrix}\begin{bmatrix}\texttt{f}\\1\end{bmatrix}\begin{bmatrix}\texttt{␣}\\1\end{bmatrix}\begin{bmatrix}\texttt{e}\\1\end{bmatrix}\begin{bmatrix}\texttt{␣}\\1\end{bmatrix}\begin{bmatrix}\texttt{l}\\3\end{bmatrix}\begin{bmatrix}\texttt{a}\\3\end{bmatrix}\begin{bmatrix}\texttt{t}\\1\end{bmatrix}\begin{bmatrix}\texttt{a}\\1\end{bmatrix}$

$\rightsquigarrow \quad r = |RL(B)| = 12; \quad n = 17$

Run-length BWT Compression

- amazingly, just run-length compressing the BWT is already powerful!
- r = number of runs in BWT

Example:

S = `alf␣eats␣alfalfa$`

B = `asff$f␣e␣lllaaata`

$RL(B) = [\mathtt{a}_1][\mathtt{s}_1][\mathtt{f}_2][\mathtt{\$}_1][\mathtt{f}_1][\mathtt{␣}_1][\mathtt{e}_1][\mathtt{␣}_1][\mathtt{l}_3][\mathtt{a}_3][\mathtt{t}_1][\mathtt{a}_1]$

$\rightsquigarrow$ $r = |RL(B)| = 12$; $n = 17$

Larger Example:

S = `have␣had␣hadnt␣hasnt␣havent␣has␣what$`

B = `tedtttshhhhhhhaavv␣␣␣␣w$␣edsaaannnaa␣`

$\rightsquigarrow$ $r = 19$; $n = 36$

- Indeed: $r = O(z \log^2(n))$, z number of LZ77 phrases proven in 2019 (!)

Kempa, Kociumaka: *Resolution of the Burrows-Wheeler Transform Conjecture*, CACM **2022**

7.6 Inverting the BWT

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Example:

$B =$ `ard$rcaaaabb`

$S =$

D

0	(a, 0)
1	(r, 1)
2	(d, 2)
3	($, 3)
4	(r, 4)
5	(c, 5)
6	(a, 6)
7	(a, 7)
8	(a, 8)
9	(a, 9)
10	(b, 10)
11	(b, 11)

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Example:

$B =$ `ard$rcaaaabb`

$S =$

	D
0	(a, 0)
1	(r, 1)
2	(d, 2)
3	($, 3)
4	(r, 4)
5	(c, 5)
6	(a, 6)
7	(a, 7)
8	(a, 8)
9	(a, 9)
10	(b, 10)
11	(b, 11)

	sorted D	
	char	next
0	($,	3)
1	(a,	0)
2	(a,	6)
3	(a,	7)
4	(a,	8)
5	(a,	9)
6	(b,	10)
7	(b,	11)
8	(c,	5)
9	(d,	2)
10	(r,	1)
11	(r,	4)

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Example:

B = `ard$rcaaaabb`

S = `a`

	D
0	(a, 0)
1	(r, 1)
2	(d, 2)
3	($, 3)
4	(r, 4)
5	(c, 5)
6	(a, 6)
7	(a, 7)
8	(a, 8)
9	(a, 9)
10	(b,10)
11	(b,11)

	sorted D	
	char	next
0	($,	3)
1	(a,	0)
2	(a,	6)
3	(a,	7)
4	(a,	8)
5	(a,	9)
6	(b,	10)
7	(b,	11)
8	(c,	5)
9	(d,	2)
10	(r,	1)
11	(r,	4)

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Example:

B = `ard$rcaaaabb`

S = `ab`

	D
0	(a, 0)
1	(r, 1)
2	(d, 2)
3	($, 3)
4	(r, 4)
5	(c, 5)
6	(a, 6)
7	(a, 7)
8	(a, 8)
9	(a, 9)
10	(b, 10)
11	(b, 11)

	sorted D
	char next
0	($, 3)
1	(a, 0)
2	(a, 6)
3	(a, 7)
4	(a, 8)
5	(a, 9)
6	(b, 10)
7	(b, 11)
8	(c, 5)
9	(d, 2)
10	(r, 1)
11	(r, 4)

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Example:

B = `ard$rcaaaabb`

S = `abr`

	D
0	(a, 0)
1	(r, 1)
2	(d, 2)
3	($, 3)
4	(r, 4)
5	(c, 5)
6	(a, 6)
7	(a, 7)
8	(a, 8)
9	(a, 9)
10	(b, 10)
11	(b, 11)

	sorted D	
	char	next
0	($,	3)
1	(a,	0)
2	(a,	6)
3	(a,	7)
4	(a,	8)
5	(a,	9)
6	(b,	10)
7	(b,	11)
8	(c,	5)
9	(d,	2)
10	(r,	1)
11	(r,	4)

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Example:

B = `ard$rcaaaabb`

S = `abra`

	D	sorted D (char, next)
0	(a, 0)	($, 3)
1	(r, 1)	(a, 0)
2	(d, 2)	(a, 6)
3	($, 3)	(a, 7)
4	(r, 4)	(a, 8)
5	(c, 5)	(a, 9)
6	(a, 6)	(b, 10)
7	(a, 7)	(b, 11)
8	(a, 8)	(c, 5)
9	(a, 9)	(d, 2)
10	(b, 10)	(r, 1)
11	(b, 11)	(r, 4)

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Example:

B = `ard$rcaaaabb`

S = `abrac`

	D	sorted D (char, next)
0	(a, 0)	($, 3)
1	(r, 1)	(a, 0)
2	(d, 2)	(a, 6)
3	($, 3)	(a, 7)
4	(r, 4)	(a, 8)
5	(c, 5)	(a, 9)
6	(a, 6)	(b, 10)
7	(a, 7)	(b, 11)
8	(a, 8)	(c, 5)
9	(a, 9)	(d, 2)
10	(b, 10)	(r, 1)
11	(b, 11)	(r, 4)

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Example:

B = `ard$rcaaaabb`

S = `abraca`

	D
0	(a, 0)
1	(r, 1)
2	(d, 2)
3	($, 3)
4	(r, 4)
5	(c, 5)
6	(a, 6)
7	(a, 7)
8	(a, 8)
9	(a, 9)
10	(b, 10)
11	(b, 11)

	sorted D	
	char	next
0	($,	3)
1	(a,	0)
2	(a,	6)
3	(a,	7)
4	(a,	8)
5	(a,	9)
6	(b,	10)
7	(b,	11)
8	(c,	5)
9	(d,	2)
10	(r,	1)
11	(r,	4)

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Example:

B = ard$rcaaaabb

S = abracad

	D
0	(a, 0)
1	(r, 1)
2	(d, 2)
3	($, 3)
4	(r, 4)
5	(c, 5)
6	(a, 6)
7	(a, 7)
8	(a, 8)
9	(a, 9)
10	(b, 10)
11	(b, 11)

	sorted D	
	char	next
0	($,	3)
1	(a,	0)
2	(a,	6)
3	(a,	7)
4	(a,	8)
5	(a,	9)
6	(b,	10)
7	(b,	11)
8	(c,	5)
9	(d,	2)
10	(r,	1)
11	(r,	4)

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Example:

B = `ard$rcaaaabb`

S = `abracada`

	D
0	(a, 0)
1	(r, 1)
2	(d, 2)
3	($, 3)
4	(r, 4)
5	(c, 5)
6	(a, 6)
7	(a, 7)
8	(a, 8)
9	(a, 9)
10	(b, 10)
11	(b, 11)

	sorted D	
	char	next
0	($,	3)
1	(a,	0)
2	(a,	6)
3	(a,	7)
4	(a,	8)
5	(a,	9)
6	(b,	10)
7	(b,	11)
8	(c,	5)
9	(d,	2)
10	(r,	1)
11	(r,	4)

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Example:

B = `ard$rcaaaabb`

S = `abracadab`

	D	sorted D (char, next)
0	(a, 0)	($, 3)
1	(r, 1)	(a, 0)
2	(d, 2)	(a, 6)
3	($, 3)	(a, 7)
4	(r, 4)	(a, 8)
5	(c, 5)	(a, 9)
6	(a, 6)	(b, 10)
7	(a, 7)	(b, 11)
8	(a, 8)	(c, 5)
9	(a, 9)	(d, 2)
10	(b, 10)	(r, 1)
11	(b, 11)	(r, 4)

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Example:

B = `ard$rcaaaabb`

S = `abracadabr`

	D
0	(a, 0)
1	(r, 1)
2	(d, 2)
3	($, 3)
4	(r, 4)
5	(c, 5)
6	(a, 6)
7	(a, 7)
8	(a, 8)
9	(a, 9)
10	(b, 10)
11	(b, 11)

	sorted D	
	char	next
0	($,	3)
1	(a,	0)
2	(a,	6)
3	(a,	7)
4	(a,	8)
5	(a,	9)
6	(b,	10)
7	(b,	11)
8	(c,	5)
9	(d,	2)
10	(r,	1)
11	(r,	4)

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Example:

B = `ard$rcaaaabb`

S = `abracadabra`

	D
0	(a, 0)
1	(r, 1)
2	(d, 2)
3	($, 3)
4	(r, 4)
5	(c, 5)
6	(a, 6)
7	(a, 7)
8	(a, 8)
9	(a, 9)
10	(b, 10)
11	(b, 11)

	sorted D	
	char	next
0	($,	3)
1	(a,	0)
2	(a,	6)
3	(a,	7)
4	(a,	8)
5	(a,	9)
6	(b,	10)
7	(b,	11)
8	(c,	5)
9	(d,	2)
10	(r,	1)
11	(r,	4)

Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But can we get T back?*

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort D *stably* with respect to *first entry*.
  3. Use D as linked list with (char, next entry)

Example:

B = `ard$rcaaaabb`

S = `abracadabra$`

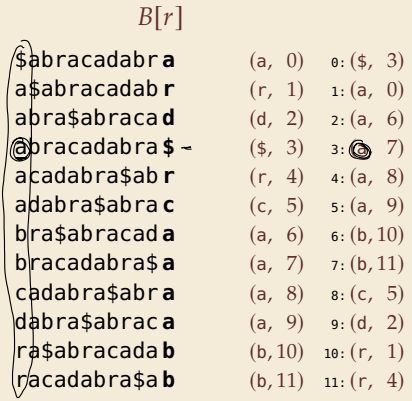
	D	sorted D (char, next)
0	(a, 0)	($, 3)
1	(r, 1)	(a, 0)
2	(d, 2)	(a, 6)
3	($, 3)	(a, 7)
4	(r, 4)	(a, 8)
5	(c, 5)	(a, 9)
6	(a, 6)	(b, 10)
7	(a, 7)	(b, 11)
8	(a, 8)	(c, 5)
9	(a, 9)	(d, 2)
10	(b, 10)	(r, 1)
11	(b, 11)	(r, 4)

Inverse BWT – The magic revealed

- Inverse BWT very easy to compute:
  - only sort individual characters in B (not suffixes)
  - $\rightsquigarrow$ $O(n)$ with counting sort
- *but why does this work!?*

Inverse BWT – The magic revealed

- Inverse BWT very easy to compute:
  - only sort individual characters in B (not suffixes)
  - $\rightsquigarrow$ $O(n)$ with counting sort
- *but why does this work!?*
- decode char by char
  - can find unique $ $\rightsquigarrow$ starting row



Inverse BWT – The magic revealed

- Inverse BWT very easy to compute:
  - only sort individual characters in B (not suffixes)
  - $\rightsquigarrow$ $O(n)$ with counting sort
- *but why does this work!?*
- decode char by char
  - can find unique `$` $\rightsquigarrow$ starting row
- to get next char, we need
  - (i) char in *first* column of *current row*
  - (ii) find row with that char's copy in BWT
  - $\rightsquigarrow$ then we can walk through and decode

$B[r]$		
$abracadabra	(a, 0)	0: ($, 3)
a$abracadabr	(r, 1)	1: (a, 0)
abra$abracad	(d, 2)	2: (a, 6)
@bracadabra$	($, 3)	3: (a, 7)
acadabra$abr	(r, 4)	4: (a, 8)
adabra$abrac	(c, 5)	5: (a, 9)
bra$abracada	(a, 6)	6: (b, 10)
bracadabra$a	(a, 7)	7: (b, 11)
cadabra$abra	(a, 8)	8: (c, 5)
dabra$abraca	(a, 9)	9: (d, 2)
ra$abracadab	(b, 10)	10: (r, 1)
racadabra$ab	(b, 11)	11: (r, 4)

Inverse BWT – The magic revealed

- Inverse BWT very easy to compute:
  - only sort individual characters in B (not suffixes)
  - $\rightsquigarrow$ $O(n)$ with counting sort
- *but why does this work!?*
- decode char by char
  - can find unique `$` $\rightsquigarrow$ starting row
- to get next char, we need
  - (i) char in *first* column of *current row*
  - (ii) find row with that char's copy in BWT
  - $\rightsquigarrow$ then we can walk through and decode
- for (i): first col = chars of B in sorted order ✓

$B[r]$		
$abracadabra	(a, 0)	0: ($, 3)
a$abracadabr	(r, 1)	1: (a, 0)
abra$abracad	(d, 2)	2: (a, 6)
abracadabra$	($, 3)	3: (a, 7)
acadabra$abr	(r, 4)	4: (a, 8)
adabra$abrac	(c, 5)	5: (a, 9)
bra$abracada	(a, 6)	6: (b, 10)
bracadabra$a	(a, 7)	7: (b, 11)
cadabra$abra	(a, 8)	8: (c, 5)
dabra$abraca	(a, 9)	9: (d, 2)
ra$abracadab	(b, 10)	10: (r, 1)
racadabra$ab	(b, 11)	11: (r, 4)

Inverse BWT – The magic revealed

- Inverse BWT very easy to compute:
  - only sort individual characters in B (not suffixes)
  - $\rightsquigarrow$ $O(n)$ with counting sort
- *but why does this work!?*
- decode char by char
  - can find unique $ $\rightsquigarrow$ starting row
- to get next char, we need
  - (i) char in *first* column of *current row*
  - (ii) find row with that char's copy in BWT
  - $\rightsquigarrow$ then we can walk through and decode
- for (i): first col = chars of B in sorted order ✓
- for (ii): relative order of same character stays same:
  ith a in first column = ith a in BWT

$B[r]$		
$abracadabra	(a, 0)	0: ($, 3)
a$abracadabr	(r, 1)	1: (a, 0)
abra$abracad	(d, 2)	2: (a, 6)
abracadabra$	($, 3)	3: (a, 7)
acadabra$abr	(r, 4)	4: (a, 8)
adabra$abrac	(c, 5)	5: (a, 9)
bra$abracada	(a, 6)	6: (b, 10)
bracadabra$a	(a, 7)	7: (b, 11)
cadabra$abra	(a, 8)	8: (c, 5)
dabra$abraca	(a, 9)	9: (d, 2)
ra$abracadab	(b, 10)	10: (r, 1)
racadabra$ab	(b, 11)	11: (r, 4)

Inverse BWT – The magic revealed

- Inverse BWT very easy to compute:
  - only sort individual characters in B (not suffixes)
  - $\rightsquigarrow$ $O(n)$ with counting sort
- *but why does this work!?*
- decode char by char
  - can find unique $ $\rightsquigarrow$ starting row
- to get next char, we need
  - (i) char in *first* column of *current row*
  - (ii) find row with that char's copy in BWT
  - $\rightsquigarrow$ then we can walk through and decode
- for (i): first col = chars of B in sorted order ✓
- for (ii): relative order of same character stays same:
  ith a in first column = ith a in BWT

$B[r]$		
$abracadabra	(a, 0)	0: ($, 3)
a$abracadabr	(r, 1)	1: (a, 0)
abra$abracad	(d, 2)	2: (a, 6)
abracadabra$	($, 3)	3: (a, 7)
acadabra$abr	(r, 4)	4: (a, 8)
adabra$abrac	(c, 5)	5: (a, 9)
bra$abracada	(a, 6)	6: (b, 10)
bracadabra$a	(a, 7)	7: (b, 11)
cadabra$abra	(a, 8)	8: (c, 5)
dabra$abraca	(a, 9)	9: (d, 2)
ra$abracadab	(b, 10)	10: (r, 1)
racadabra$ab	(b, 11)	11: (r, 4)

Inverse BWT – The magic revealed

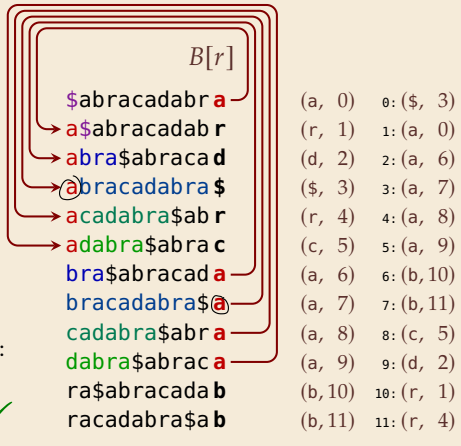
- Inverse BWT very easy to compute:
  - only sort individual characters in B (not suffixes)
  - $\rightsquigarrow$ $O(n)$ with counting sort
- *but why does this work!?*
- decode char by char
  - can find unique $ $\rightsquigarrow$ starting row
- to get next char, we need
  - (i) char in *first* column of *current row*
  - (ii) find row with that char's copy in BWT
  - $\rightsquigarrow$ then we can walk through and decode
- for (i): first col = chars of B in sorted order ✓
- for (ii): relative order of same character stays same:
  ith a in first column = ith a in BWT
  - $\rightsquigarrow$ stably sorting $(B[r], r)$ by first entry enough

	$B[r]$		
0:	$abracadabra	(a, 0)	0: ($, 3)
1:	a$abracadabr	(r, 1)	1: (a, 0)
2:	abra$abracad	(d, 2)	2: (a, 6)
3:	abracadabra$	($, 3)	3: (a, 7)
4:	acadabra$abr	(r, 4)	4: (a, 8)
5:	adabra$abrac	(c, 5)	5: (a, 9)
6:	bra$abracada	(a, 6)	6: (b, 10)
7:	bracadabra$a	(a, 7)	7: (b, 11)
8:	cadabra$abra	(a, 8)	8: (c, 5)
9:	dabra$abraca	(a, 9)	9: (d, 2)
10:	ra$abracadab	(b, 10)	10: (r, 1)
11:	racadabra$ab	(b, 11)	11: (r, 4)

Inverse BWT – The magic revealed

- Inverse BWT very easy to compute:
  - only sort individual characters in B (not suffixes)
  - $\rightsquigarrow$ $O(n)$ with counting sort

- *but why does this work!?*
- decode char by char
  - can find unique $ $\rightsquigarrow$ starting row
- to get next char, we need
  - (i) char in *first* column of *current row*
  - (ii) find row with that char's copy in BWT
  - $\rightsquigarrow$ then we can walk through and decode
- for (i): first col = chars of B in sorted order ✓
- for (ii): relative order of same character stays same:
  ith a in first column = ith a in BWT
  - $\rightsquigarrow$ stably sorting $(B[r], r)$ by first entry enough ✓



Random Access Decoding

Can similarly output **any substring** $T[i..i+\ell)$ if we know inverse suffix array:
Simply do ℓ steps of the inverse BWT starting at $r = R[i-1]$!

i	$R[i]$	T_i	r	$L[r]$	$T_{L[r]}$	$B[r]$	D	sort(D)
0	6th	bananaban$	0	9	$bananaba	n	(n,0)	0: ($,6)
1	4th	ananaban$b	1	5	aban$banan	n	(n,1)	1: (a,5)
2	9th	nanaban$ba	2	7	an$banana	b	(b,2)	2: (a,7)
3	3rd	anaban$ban	3	3	anaban$ba	n	(n,3)	3: (a,8)
4	8th	naban$bana	4	1	ananaban$	b	(b,4)	4: (a,9)
5	1st	aban$banan	5	6	ban$banan	a	(a,5)	5: (b,2)
6	5th	ban$banana	6	0	bananaban	$	($,6)	6: (b,4)
7	2nd	an$bananab	7	8	n$bananab	a	(a,7)	7: (n,0)
8	7th	n$bananaba	8	4	naban$ban	a	(a,8)	8: (n,1)
9	0th	$bananaban	9	2	nanaban$b	a	(a,9)	9: (n,3)

sort suffixes

T[5..8)

aba

Random Access Decoding

Can similarly output **any substring** $T[i..i+\ell)$ if we know inverse suffix array:
Simply do ℓ steps of the inverse BWT starting at $r = R[i-1]$!

i	$R[i]$	T_i	r	$L[r]$	$T_{L[r]}$	$B[r]$	D	sort(D)
0	6th	bananaban$	0	9	$bananaba	**n**	(n,0)	0: ($,6)
1	4th	ananaban$b	1	5	aban$bana	**n**	(n,1)	1: (a,5)
2	9th	nanaban$ba	2	7	an$banana	**b**	(b,2)	2: (a,7)
3	3th	anaban$ban	3	3	anaban$ba	**n**	(n,3)	3: (a,8)
4	8th	naban$bana	4	1	ananaban$	**b**	(b,4)	4: (a,9)
5	1th	aban$banan	5	6	ban$banan	**a**	(a,5)	5: (b,2)
6	5th	ban$banana	6	0	bananaban	**$**	($,6)	6: (b,4)
7	2th	an$bananab	7	8	n$bananab	**a**	(a,7)	7: (n,0)
8	7th	n$bananaba	8	4	naban$ban	**a**	(a,8)	8: (n,1)
9	0th	$bananaban	9	2	nanaban$b	**a**	(a,9)	9: (n,3)

sort suffixes

Decoding only needs access to

1. ith char c of sort(T) = sort(B)
2. *position* of (that copy of) c in B

⇝ *If we have that, can skip sorting / storing all of D!*

7.7 Random Access in BWT

Rank & Select on Sequences

Recall: Decoding only needs access to

1. ith char c of $\text{sort}(T) = \text{sort}(B)$
2. *position* of (that copy of) c in B

Both can be supported using rank/select on sequences.

- $\text{rank}_c(T[0..n), i) = |T[0..i)|_c$
  $= \#c$ in first i characters of T
- $\text{select}_c(T[0..n), r)$
  $= \min\{j : |T[0..j]|_c \geq r\} \cup \{n\}$
  $= $ index of rth c in T, $(r = 1, 2, \ldots)$

#occurrences of c

$T[0..9)$

0	1	2	3	4	5	6	7	8
b	a	n	a	n	a	b	a	n

	0	1	2	3	4	5	6	7	8	9
$\text{rank}_a(T, i)$	0	0	1	1	2	2	3	3	4	4
$\text{rank}_b(T, i)$	0	1	1	1	1	1	1	2	2	2
$\text{rank}_n(T, i)$	0	0	0	1	1	2	2	2	2	3

	/	1	2	3	4	5	6	7	8	9
$\text{select}_a(T, r)$	/	1	3	5	7	9	9	9	9	9
$\text{select}_b(T, r)$	/	0	6	9	9	9	9	9	9	9
$\text{select}_n(T, r)$	/	2	4	8	9	9	9	9	9	9

$\text{sort}(T)$

0	1	2	3	4	5	6	7	8
a	a	a	a	b	b	n	n	n

$O[0..\sigma] = [0, 4, 6, 9]$

Random Access in BWT

- store offsets $O[c] = \sum_{c'=0}^{c-1} |B|_{c'}$ for $c \in \Sigma$
- ith char of $\text{sort}(B)$ = unique c for which $O[c] \leq i < O[c+1]$
- position of rth c in B = $\text{select}_c(B, r)$

Wavelet Trees

The **Wavelet Trees** for a $T \in [0..n)$ over $\Sigma = [0..\sigma)$

- supports access to $T[i]$ in $O(\log \sigma)$ time,
- $\text{rank}_c(T, i)$ and $\text{select}_c(T, r)$ in $O(\log \sigma)$ time, and
- occupies $\sim n \lg \sigma$ *bits* of space.

Wavelet Trees

The **Wavelet Trees** for a $T \in [0..n)$ over $\Sigma = [0..\sigma)$

- supports access to $T[i]$ in $O(\log \sigma)$ time,
- $\mathrm{rank}_c(T, i)$ and $\mathrm{select}_c(T, r)$ in $O(\log \sigma)$ time, and
- occupies $\sim n \lg \sigma$ *bits* of space. *(Further compression possible!)* $\rightsquigarrow$ *Advanced Data Structures*

Wavelet Trees

The **Wavelet Trees** for a $T \in [0..n)$ over $\Sigma = [0..\sigma)$

- supports access to $T[i]$ in $O(\log \sigma)$ time,
- $\text{rank}_c(T, i)$ and $\text{select}_c(T, r)$ in $O(\log \sigma)$ time, and
- occupies $\sim n \lg \sigma$ *bits* of space. *(Further compression possible!)* $\rightsquigarrow$ *Advanced Data Structures*
- The generalized σ-$\text{rank}_c(T, i) = \text{rank}_c(T, i) + \sum_{c' < c} |T|_{c'}$ is also supported in $O(\log \sigma)$ time

Wavelet Trees

The **Wavelet Trees** for a $T \in [0..n)$ over $\Sigma = [0..\sigma)$

- supports access to $T[i]$ in $O(\log \sigma)$ time,
- $\text{rank}_c(T, i)$ and $\text{select}_c(T, r)$ in $O(\log \sigma)$ time, and
- occupies $\sim n \lg \sigma$ *bits* of space. *(Further compression possible!)* $\rightsquigarrow$ *Advanced Data Structures*
- The generalized $\sigma\text{-rank}_c(T, i) = \text{rank}_c(T, i) + \sum_{c' < c} |T|_{c'}$ is also supported in $O(\log \sigma)$ time

Storing $B[0..n]$ as a wavelet tree $\rightsquigarrow$ reconstruct ℓ chars from T in $O(\ell \log \sigma)$ time

Wavelet Trees

The **Wavelet Trees** for a $T \in [0..n)$ over $\Sigma = [0..\sigma)$

- supports access to $T[i]$ in $O(\log \sigma)$ time,
- $\text{rank}_c(T, i)$ and $\text{select}_c(T, r)$ in $O(\log \sigma)$ time, and
- occupies $\sim n \lg \sigma$ *bits* of space. *(Further compression possible!)* $\rightsquigarrow$ *Advanced Data Structures*
- The generalized $\sigma\text{-rank}_c(T, i) = \text{rank}_c(T, i) + \sum_{c' < c} |T|_{c'}$ is also supported in $O(\log \sigma)$ time

Storing $B[0..n]$ as a wavelet tree $\rightsquigarrow$ reconstruct ℓ chars from T in $O(\ell \log \sigma)$ time
if starting position known

Storing **every tth entry** of $R[0..n]$ $\rightsquigarrow$ may need to go back t characters for access
$\rightsquigarrow$ $O((\ell + t) \log \sigma)$ time for decode
using $\sim n \lg n/t$ extra bits of space

e. g., $t = \lg n$

Wavelet Trees

The **Wavelet Trees** for a $T \in [0..n)$ over $\Sigma = [0..\sigma)$

- supports access to $T[i]$ in $O(\log \sigma)$ time,
- $\text{rank}_c(T, i)$ and $\text{select}_c(T, r)$ in $O(\log \sigma)$ time, and
- occupies $\sim n \lg \sigma$ *bits* of space. *(Further compression possible!)* $\rightsquigarrow$ *Advanced Data Structures*
- The generalized σ-$\text{rank}_c(T, i) = \text{rank}_c(T, i) + \sum_{c' < c} |T|_{c'}$ is also supported in $O(\log \sigma)$ time

Storing $B[0..n]$ as a wavelet tree $\rightsquigarrow$ reconstruct ℓ chars from T in $O(\ell \log \sigma)$ time
if starting position known

Storing **every tth entry** of $R[0..n]$ $\rightsquigarrow$ may need to go back t characters for access
$\rightsquigarrow$ $O((\ell + t) \log \sigma)$ time for decode
using $\sim n \lg n/t$ extra bits of space

e. g., $t = \lg n$

Locally decodable BWT

- no longer need to store $T[0..n)$!
- compressible (e. g., Wavelet trees with compressed bitvectors)

7.8 Searching in the BWT

Backwards Search

*Recall how the sorted suffixes in a suffix array $L[0..n]$ made **string matching** very easy.*

- Simply binary search the pattern $P[0..m)$ in L!
- $\rightsquigarrow$ all occurrences must form interval

Backwards Search

*Recall how the sorted suffixes in a suffix array $L[0..n]$ made **string matching** very easy.*

- Simply binary search the pattern $P[0..m)$ in L!

$\rightsquigarrow$ all occurrences must form interval

*With wavelet tree BWT, we can replace binary search by **backwards radix search**!*

- use $\mathrm{sort}(B)$ to locate interval for **last** character $P[m-1]$
- use one step of inverse BWT to narrow down on $P[m-2..m)$, repeat.

Backwards Search

*Recall how the sorted suffixes in a suffix array $L[0..n]$ made **string matching** very easy.*

- Simply binary search the pattern $P[0..m)$ in L!

⇝ all occurrences must form interval

*With wavelet tree BWT, we can replace binary search by **backwards radix search**!*

- use $\text{sort}(B)$ to locate interval for **last** character $P[m-1]$
- use one step of inverse BWT to narrow down on $P[m-2..m)$, repeat.

i	$R[i]$	T_i
0	6^{th}	bananaban$
1	4^{th}	ananaban$b
2	9^{th}	nanaban$ba
3	3^{th}	anaban$ban
4	8^{th}	naban$bana
5	1^{th}	aban$banan
6	5^{th}	ban$banana
7	2^{th}	an$bananab
8	7^{th}	n$bananaba
9	0^{th}	$bananaban

P = ana

$L[r]$	r	$T_{L[r]}$	$B[r]$	sort(D)
9	0	$bananaba	**n**	0: ($, 6)
5	1	aban$bana	**n**	1: (a, 5)
7	2	an$banana	**b**	2: (a, 7)
3	3	anaban$ba	**n**	3: (a, 8)
1	4	ananaban$	**b**	4: (a, 9)
6	5	ban$banan	**a**	5: (b, 2)
0	6	bananaban	**$**	6: (b, 4)
8	7	n$bananab	**a**	7: (n, 0)
4	8	naban$ban	**a**	8: (n, 1)
2	9	nanaban$b	**a**	9: (n, 3)

Backwards Search

*Recall how the sorted suffixes in a suffix array $L[0..n]$ made **string matching** very easy.*

- Simply binary search the pattern $P[0..m)$ in L!

⇝ all occurrences must form interval

*With wavelet tree BWT, we can replace binary search by **backwards radix search**!*

- use $\mathrm{sort}(B)$ to locate interval for **last** character $P[m-1]$
- use one step of inverse BWT to narrow down on $P[m-2..m)$, repeat.

i	$R[i]$	T_i
0	6^{th}	bananaban$
1	4^{th}	ananaban$b
2	9^{th}	nanaban$ba
3	3^{th}	anaban$ban
4	8^{th}	naban$bana
5	1^{th}	aban$banan
6	5^{th}	ban$banana
7	2^{th}	an$bananab
8	7^{th}	n$bananaba
9	0^{th}	$bananaban

P = ana

$L[r]$	r	$T_{L[r]}$	$B[r]$	sort(D)
9	0	$bananaba	**n**	0: ($, 6)
5	1	aban$bana	**n**	1: (a, 5)
7	2	an$banana	**b**	2: (a, 7)
3	3	anaban$ba	**n**	3: (a, 8)
1	4	ananaban$	**b**	4: (a, 9)
6	5	ban$banan	**a**	5: (b, 2)
0	6	bananaban	**$**	6: (b, 4)
8	7	n$bananab	**a**	7: (n, 0)
4	8	naban$ban	**a**	8: (n, 1)
2	9	nanaban$b	**a**	9: (n, 3)

Backwards Search

*Recall how the sorted suffixes in a suffix array $L[0..n]$ made **string matching** very easy.*

- Simply binary search the pattern $P[0..m)$ in L!

⇝ all occurrences must form interval

*With wavelet tree BWT, we can replace binary search by **backwards radix search**!*

- use $\mathrm{sort}(B)$ to locate interval for **last** character $P[m-1]$
- use one step of inverse BWT to narrow down on $P[m-2..m)$, repeat.

i	$R[i]$	T_i
0	6^{th}	bananaban$
1	4^{th}	ananaban$b
2	9^{th}	nanaban$ba
3	3^{th}	anaban$ban
4	8^{th}	naban$bana
5	1^{th}	aban$banan
6	5^{th}	ban$banana
7	2^{th}	an$bananab
8	7^{th}	n$bananaba
9	0^{th}	$bananaban

P = ana

$L[r]$	r	$T_{L[r]}$	$B[r]$	sort(D)
9	0	$bananaba	**n**	0: ($, 6)
5	1	aban$bana	**n**	1: (a, 5)
7	2	an$banana	**b**	2: (a, 7)
3	3	anaban$ba	**n**	3: (a, 8)
1	4	ananaban$	**b**	4: (a, 9)
6	5	ban$banan	**a**	5: (b, 2)
0	6	bananaban	**$**	6: (b, 4)
8	7	n$bananab	**a**	7: (n, 0)
4	8	naban$ban	**a**	8: (n, 1)
2	9	nanaban$b	**a**	9: (n, 3)

Backwards Search

*Recall how the sorted suffixes in a suffix array $L[0..n]$ made **string matching** very easy.*

- Simply binary search the pattern $P[0..m)$ in L!

⇝ all occurrences must form interval

*With wavelet tree BWT, we can replace binary search by **backwards radix search**!*

- use $\mathrm{sort}(B)$ to locate interval for **last** character $P[m-1]$
- use one step of inverse BWT to narrow down on $P[m-2..m)$, repeat.

i	$R[i]$	T_i
0	6^{th}	bananaban$
1	4^{th}	ananaban$b
2	9^{th}	nanaban$ba
3	3^{th}	anaban$ban
4	8^{th}	naban$bana
5	1^{th}	aban$banan
6	5^{th}	ban$banana
7	2^{th}	an$bananab
8	7^{th}	n$bananaba
9	0^{th}	$bananaban

P = ana

$L[r]$	r	$T_{L[r]}$	$B[r]$	sort(D)
9	0	$bananaba	n	0: ($, 6)
5	1	aban$bana	n	1: (a, 5)
7	2	an$banana	b	2: (a, 7)
3	3	anaban$ba	n	3: (a, 8)
1	4	ananaban$	b	4: (a, 9)
6	5	ban$banan	a	5: (b, 2)
0	6	bananaban	$	6: (b, 4)
8	7	n$bananab	a	7: (n, 0)
4	8	naban$ban	a	8: (n, 1)
2	9	nanaban$b	a	9: (n, 3)

Backwards Search

*Recall how the sorted suffixes in a suffix array $L[0..n]$ made **string matching** very easy.*

- Simply binary search the pattern $P[0..m)$ in L!
- ⇝ all occurrences must form interval

*With wavelet tree BWT, we can replace binary search by **backwards radix search**!*

- use $\mathrm{sort}(B)$ to locate interval for **last** character $P[m-1]$
- use one step of inverse BWT to narrow down on $P[m-2..m)$, repeat.

i	$R[i]$	T_i
0	6^{th}	bananaban$
1	4^{th}	ananaban$b
2	9^{th}	nanaban$ba
3	3^{th}	anaban$ban
4	8^{th}	naban$bana
5	1^{th}	aban$banan
6	5^{th}	ban$banana
7	2^{th}	an$bananab
8	7^{th}	n$bananaba
9	0^{th}	$bananaban

P = ana

$L[r]$	r	$T_{L[r]}$	$B[r]$	sort(D)
9	0	$bananaba	**n**	0: ($, 6)
5	1	aban$bana	**n**	1: (a, 5)
7	2	an$banana	**b**	2: (a, 7)
3	3	anaban$ba	**n**	3: (a, 8)
1	4	ananaban$	**b**	4: (a, 9)
6	5	ban$banan	**a**	5: (b, 2)
0	6	bananaban	**$**	6: (b, 4)
8	7	n$bananab	**a**	7: (n, 0)
4	8	naban$ban	**a**	8: (n, 1)
2	9	nanaban$b	**a**	9: (n, 3)

Backwards Search

*Recall how the sorted suffixes in a suffix array $L[0..n]$ made **string matching** very easy.*

- Simply binary search the pattern $P[0..m)$ in L!
- ⇝ all occurrences must form interval

*With wavelet tree BWT, we can replace binary search by **backwards radix search**!*

- use $\mathrm{sort}(B)$ to locate interval for **last** character $P[m-1]$
- use one step of inverse BWT to narrow down on $P[m-2..m)$, repeat.

i	$R[i]$	T_i
0	6^{th}	bananaban$
1	4^{th}	ananaban$b
2	9^{th}	nanaban$ba
3	3^{th}	anaban$ban
4	8^{th}	naban$bana
5	1^{th}	aban$banan
6	5^{th}	ban$banana
7	2^{th}	an$bananab
8	7^{th}	n$bananaba
9	0^{th}	$bananaban

P = ana

$L[r]$	r	$T_{L[r]}$	$B[r]$	sort(D)
9	0	$bananaba	**n**	0: ($, 6)
5	1	aban$bana	**n**	1: (a, 5)
7	2	an$banana	**b**	2: (a, 7)
3	3	⇜ anaban$ba	**n**	3: (a, 8)
1	4	⇝ ananaban$	**b**	4: (a, 9)
6	5	ban$banan	**a**	5: (b, 2)
0	6	bananaban	**$**	6: (b, 4)
8	7	n$bananab	**a**	7: (n, 0)
4	8	naban$ban	**a**	8: (n, 1)
2	9	nanaban$b	**a**	9: (n, 3)

Backwards Search

*Recall how the sorted suffixes in a suffix array $L[0..n]$ made **string matching** very easy.*

- Simply binary search the pattern $P[0..m)$ in L!

⇝ all occurrences must form interval

*With wavelet tree BWT, we can replace binary search by **backwards radix search**!*

- use $\mathrm{sort}(B)$ to locate interval for **last** character $P[m-1]$
- use one step of inverse BWT to narrow down on $P[m-2..m)$, repeat.

i	$R[i]$	T_i
0	6^{th}	bananaban$
1	4^{th}	ananaban$b
2	9^{th}	nanaban$ba
3	3^{th}	anaban$ban
4	8^{th}	naban$bana
5	1^{th}	aban$banan
6	5^{th}	ban$banana
7	2^{th}	an$bananab
8	7^{th}	n$bananaba
9	0^{th}	$bananaban

$P = \texttt{ana}$

$L[r]$	r	$T_{L[r]}$	$B[r]$	$\mathrm{sort}(D)$
9	0	$bananaba	**n**	0: ($, 6)
5	1	aban$bana	**n**	1: (a, 5)
7	2	an$banana	**b**	2: (a, 7)
3	3	anaban$ba	**n**	3: (a, 8)
1	4	ananaban$	**b**	4: (a, 9)
6	5	ban$banan	**a**	5: (b, 2)
0	6	bananaban	**$**	6: (b, 4)
8	7	n$bananab	**a**	7: (n, 0)
4	8	naban$ban	**a**	8: (n, 1)
2	9	nanaban$b	**a**	9: (n, 3)

Backwards Search

*Recall how the sorted suffixes in a suffix array $L[0..n]$ made **string matching** very easy.*

- Simply binary search the pattern $P[0..m)$ in L!
- ⇝ all occurrences must form interval

*With wavelet tree BWT, we can replace binary search by **backwards radix search**!*

- use $\mathrm{sort}(B)$ to locate interval for **last** character $P[m-1]$
- use one step of inverse BWT to narrow down on $P[m-2..m)$, repeat.

i	$R[i]$	T_i
0	6^{th}	bananaban$
1	4^{th}	ananaban$b
2	9^{th}	nanaban$ba
3	3^{th}	anaban$ban
4	8^{th}	naban$bana
5	1^{th}	aban$banan
6	5^{th}	ban$banana
7	2^{th}	an$bananab
8	7^{th}	n$bananaba
9	0^{th}	$bananaban

P = ana

$L[r]$	r	$T_{L[r]}$	$B[r]$	sort(D)
9	0	$bananaba	n	0: ($, 6)
5	1	aban$bana	n	1: (a, 5)
7	2	an$banana	b	2: (a, 7)
3	3	anaban$ba	n	3: (a, 8)
1	4	ananaban$	b	4: (a, 9)
6	5	ban$banan	a	5: (b, 2)
0	6	bananaban	$	6: (b, 4)
8	7	n$bananab	a	7: (n, 0)
4	8	naban$ban	a	8: (n, 1)
2	9	nanaban$b	a	9: (n, 3)

Backwards Search

*Recall how the sorted suffixes in a suffix array $L[0..n]$ made **string matching** very easy.*

- Simply binary search the pattern $P[0..m)$ in L!

⇝ all occurrences must form interval

*With wavelet tree BWT, we can replace binary search by **backwards radix search**!*

- use $\mathrm{sort}(B)$ to locate interval for **last** character $P[m-1]$
- use one step of inverse BWT to narrow down on $P[m-2..m)$, repeat.

i	$R[i]$	T_i
0	6^{th}	bananaban$
1	4^{th}	ananaban$b
2	9^{th}	nanaban$ba
3	3^{th}	anaban$ban
4	8^{th}	naban$bana
5	1^{th}	aban$banan
6	5^{th}	ban$banana
7	2^{th}	an$bananab
8	7^{th}	n$bananaba
9	0^{th}	$bananaban

$P = \texttt{ana}$

$L[r]$	r	$T_{L[r]}$	$B[r]$	sort(D)
9	0	$bananaba	**n**	0: ($, 6)
5	1	aban$bana	**n**	1: (a, 5)
7	2	an$banana	**b**	2: (a, 7)
3	3	anaban$ba	**n**	3: (a, 8)
1	4	ananaban$	**b**	4: (a, 9)
6	5	ban$banan	**a**	5: (b, 2)
0	6	bananaban	**$**	6: (b, 4)
8	7	n$bananab	**a**	7: (n, 0)
4	8	naban$ban	**a**	8: (n, 1)
2	9	nanaban$b	**a**	9: (n, 3)

Backwards Search – Code

Recall total rank operation supported by wavelet trees

$$\sigma\text{-rank}_c(B, i) \;=\; \big|B[0..i)\big|_c \;+\; \sum_{c'<c} |B|_{c'}$$

```
1  procedure backwardSearch(B[0..n], P[0..m))
2      // B[0..n] given as wavelet tree
3      // returns range [s..e) of ranks for suffixes starting with P
4      c := P[m − 1]
5      s := σ-rank_c(B, 0)
6      e := σ-rank_c(B, n)
7      for j := m − 2, m − 3, . . . , 0
8          if s ≥ e break // no matches
9          c := P[j]
10         s := σ-rank_c(B, s)
11         e := σ-rank_c(B, e)
12     return [s..e)
```

Locating Matches

- Backwards Search finds interval $[s..e)$ such that

  $P[0..m) = T\big[L[r] \,..\, L[r]{+}m\big)$ *iff* $r \in [s..e)$

$\rightsquigarrow$ still need suffix array $L[0..n]$ to locate matches!

- but can detect and count occurrences even without L

Locating Matches

- Backwards Search finds interval $[s..e)$ such that

  $P[0..m) = T\big[L[r] \,..\, L[r]{+}m\big)$ *iff* $r \in [s..e)$

⇝ still need suffix array $L[0..n]$ to locate matches!

- but can detect and count occurrences even without L

Sampled Suffix Array

- As for inverse suffix array, can store $L[r]$ only for every tth starting index i in T, i. e., only store entries for ranks r with $L[r] \equiv 0 \pmod{t}$

⇝ $O(n \log n/t)$ bits of extra space

⇝ Need to continue backwards search for at most t extra characters to locate match

⇝ String matching in $O(m \log \sigma + occ \cdot t \log \sigma)$ time

Locating Matches

- Backwards Search finds interval $[s..e)$ such that

  $P[0..m) = T\big[L[r] \,..\, L[r]{+}m\big)$ *iff* $r \in [s..e)$

⇝ still need suffix array $L[0..n]$ to locate matches!

- but can detect and count occurrences even without L

Sampled Suffix Array

- As for inverse suffix array, can store $L[r]$ only for every tth starting index i in T, i. e., only store entries for ranks r with $L[r] \equiv 0 \pmod t$

⇝ $O(n \log n/t)$ bits of extra space

⇝ Need to continue backwards search for at most t extra characters to locate match

⇝ String matching in $O(m \log \sigma + occ \cdot t \log \sigma)$ time

Wavelet-tree BWT + Sampled Suffix Array = FM Index

Ferragina, Manzini: *Indexing compressed text*, JACM **2005**

FM-Index Discussion

- FM-Index is one of first *compressed self-indexes*
- can represent text using $\sim \mathcal{H}_k(T)n$ bits of space
  $\mathcal{H}_k(T)$ = kth order empirical entropy
- still widely used, e. g., as basis of *bowtie2* read alignment tool

  Langmead, Salzberg: *Fast gapped-read alignment with Bowtie 2*, Nature Methods **2012**

Ongoing research

- Reduce space for very repetitive strings (collection of genomes)

  e. g., r-index **Navarro**: *Indexing Highly Repetitive String Collections, Part II: Compressed Indexes*, ACM Comp. Surv. **2021**
- full support of suffix tree functionality with little extra space?