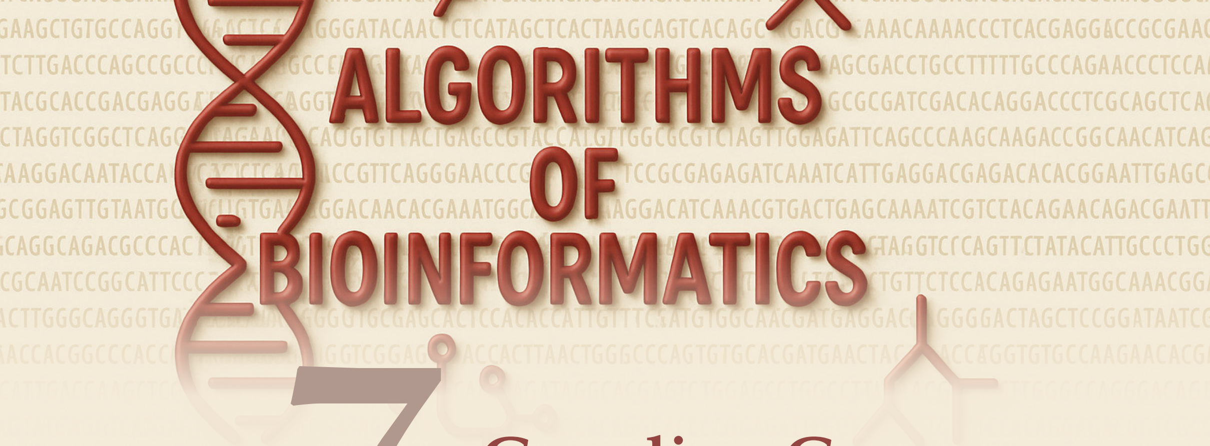


ALGORITHMS OF BIOINFORMATICS

7

Googling Genomes

15 January 2026

Prof. Dr. Sebastian Wild

CS594 (Winter 2025-26)
Philipps-Universität Marburg
version `2026-01-22 19:11` H

Outline

7 Googling Genomes

Recall Unit 6

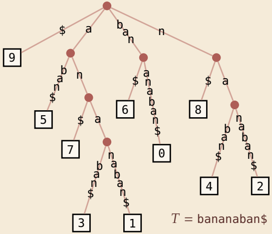
Application 4: Longest Common Extensions

- 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!



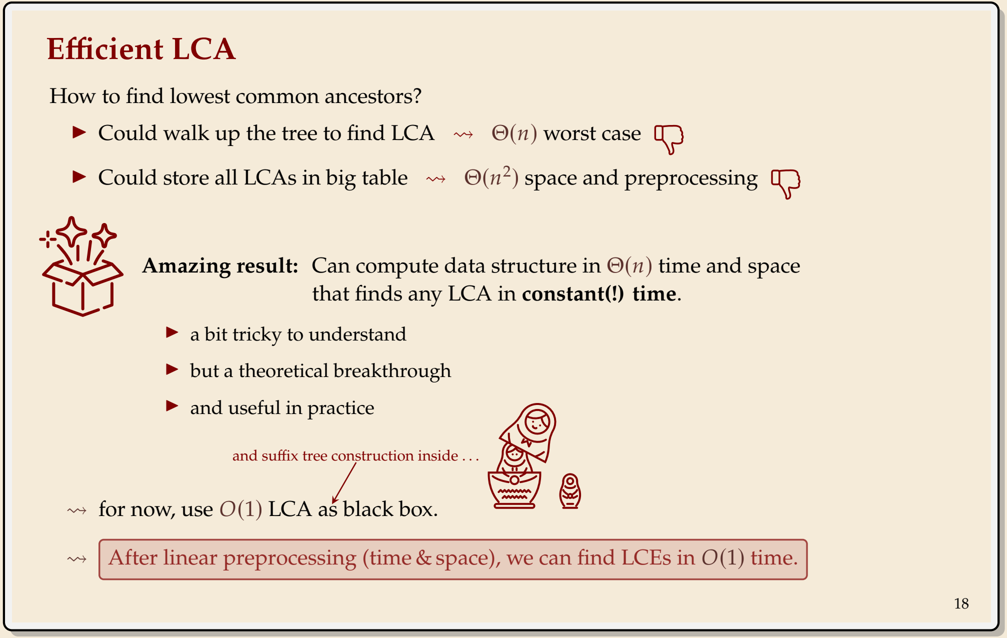
T = bananaban$

(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 ⇝ $\Theta(n)$ worst case 👎
- Could store all LCAs in big table ⇝ $\Theta(n^2)$ space and preprocessing 👎

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

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

and suffix tree construction inside . . .

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

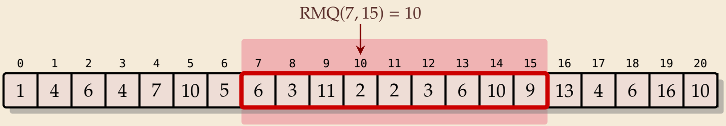
⇝ After linear preprocessing (time & space), we can find LCEs in $O(1)$ time.

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



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

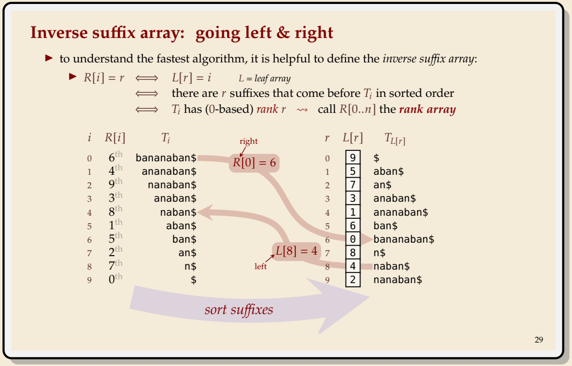
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	4	6	4	7	10	5	6	3	11	2	2	3	6	10	9	13	4	6	16	10

- **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!

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

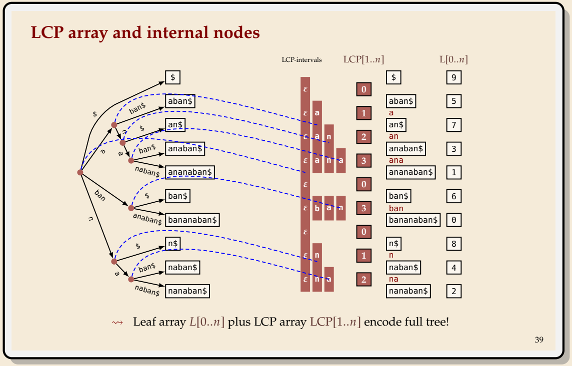


Inverse suffix array: going left & right

- 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 ⇝ 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

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

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:

  $\rightsquigarrow$ space usage $\le P(n)$

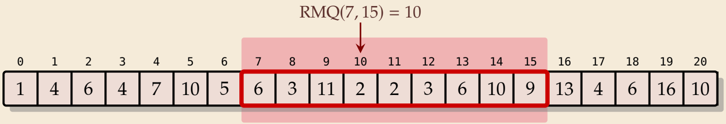
  1. **Preprocessing time**: Running time $P(n)$ of the preprocessing step
  2. **Query time**: Running time $Q(n)$ of one query (using precomputed data)
- Write $\langle P(n), Q(n) \rangle$ **time solution** for short

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



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

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

- Two easy solutions show extreme ends of scale:

1. **Scan on demand**
   - no preprocessing at all
   - answer $\mathrm{RMQ}(i, j)$ 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: $\mathrm{RMQ}(i, j) = M[i][j]$

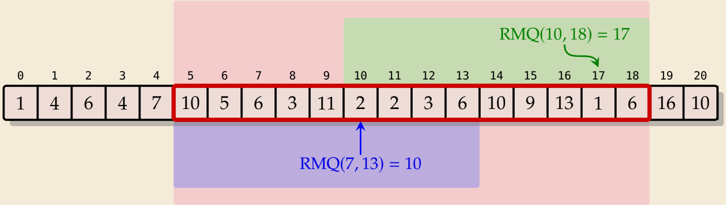
   $\rightsquigarrow \langle O(n^3), O(1) \rangle$

   - 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.
  - ⇝ $\leq n \cdot \lg n$ entries
  - ⇝ 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 $rmq_1 = \mathrm{RMQ}(i, i + 2^k - 1)$
   $rmq_2 = \mathrm{RMQ}(j - 2^k + 1, j)$

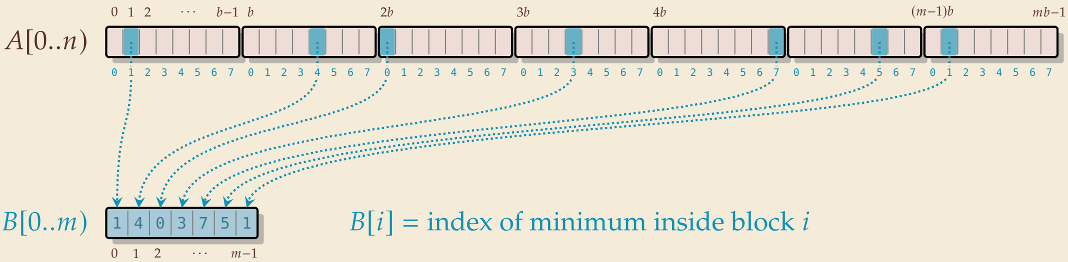
- Preprocessing can be done in $O(n \log n)$ times

⇝ $\langle O(n \log n), O(1) \rangle$ 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 $m = \lceil n/b \rceil = O(n/\log n)$



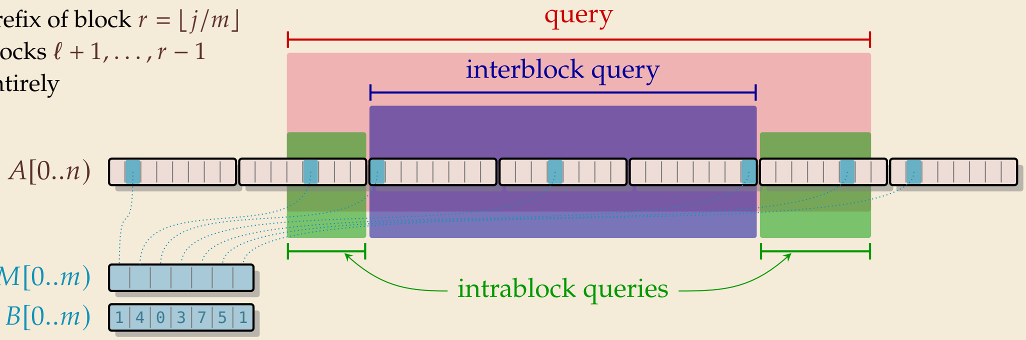
$B[i]$ = index of minimum inside block i

⇝ 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

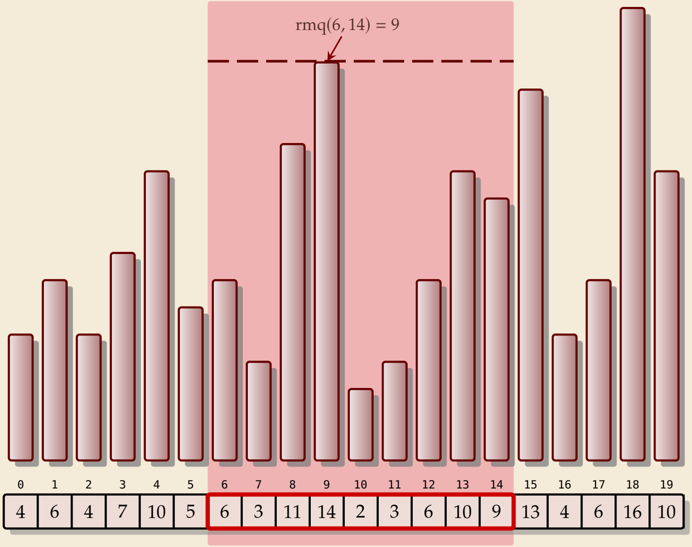


- $$\mathrm{RMQ}_\mathrm{A}(i,j) \;=\; \operatorname*{arg\,min}_{k\in K} A[k] \quad \text{with } K \;=\; \left\{ \begin{array}{l} \mathrm{RMQ}_{\text{block } \ell}\big(i-\ell b,\ (\ell+1)b-1\big), \\ b \cdot \mathrm{RMQ}_M\big(\ell+1, r-1\big) + \\ \qquad B\big[\mathrm{RMQ}_M\big(\ell+1, r-1\big)\big],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

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

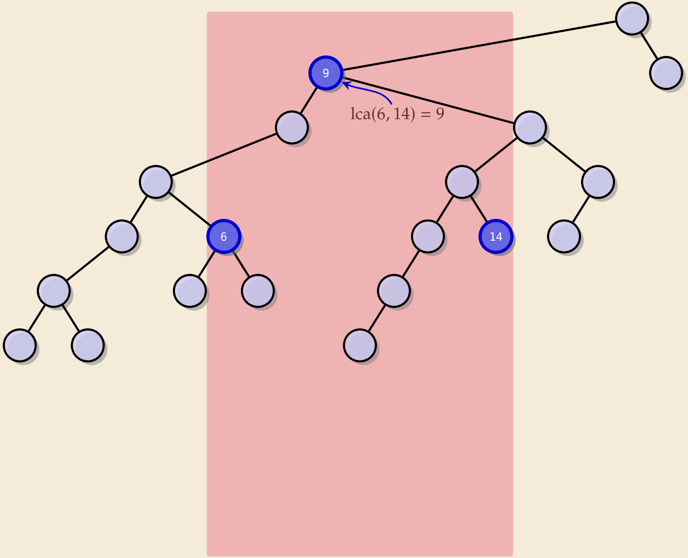
7.3 RMQ – Cartesian Trees

RMQ & LCA



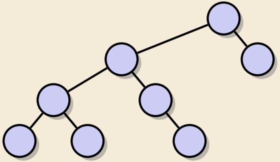
- **Range-max queries** on array A:
  $$\mathrm{rmq}_A(i,j) = \arg\max_{i \le k \le j} A[k] = \textit{index} \text{ of max}$$
- **Task:** Preprocess A,
  then answer RMQs fast
  ideally constant time!

RMQ & LCA



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

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: $\le 2^{2n}$

$\rightsquigarrow$ 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	$\mathrm{RMQ}(i,j)$
⋮			
⋮			

- $\le \sqrt{n}$ 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

$\leadsto$ $\langle O(n), O(1)\rangle$ 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: $\lg n + 2$ *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
  $T[a..n] \le_{\text{lex}} P \le_{\text{lex}} T[b..n]$
  $\ell_a = \mathrm{LCP}(T[a..n], P)$ and $\ell_b = \mathrm{LCP}(T[b..n], P)$
- avoid comparing same characters again
- Note: with RMQ on LCP array can determine $\mathrm{LCP}(T_i, T_j)$ for any $i, j \in [0..n)$

LCP Binary Search

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

  $\rightsquigarrow \ell_m = \mathrm{LCP}(T_m, P) \ge \min\{\ell_a, \ell_b\}$
- **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:** $\mathrm{LCP}(T_a, T_m) > \ell_a$
    $P >_{\text{lex}} T_m$ w/o cmps!
  - **Case 2b:** $\mathrm{LCP}(T_a, T_m) < \ell_a$
    $P <_{\text{lex}} T_m$ 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 ✓

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!

- 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

- *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
2. Sort cyclic shifts
3. Extract last column

B = `asff$f␣e␣lllaaata`

		BWT ↓
`alf␣eats␣alfalfa$`		`$alf␣eats␣alfalf`**`a`**
`lf␣eats␣alfalfa$a`		`␣alfalfa$alf␣eat`**`s`**
`f␣eats␣alfalfa$al`		`␣eats␣alfalfa$al`**`f`**
`␣eats␣alfalfa$alf`		`a$alf␣eats␣alfal`**`f`**
`eats␣alfalfa$alf␣`		`alf␣eats␣alfalfa`**`$`**
`ats␣alfalfa$alf␣e`		`alfa$alf␣eats␣al`**`f`**
`ts␣alfalfa$alf␣ea`	⇝	`alfalfa$alf␣eats`**`␣`**
`s␣alfalfa$alf␣eat`	sort	`ats␣alfalfa$alf␣`**`e`**
`␣alfalfa$alf␣eats`		`eats␣alfalfa$alf`**`␣`**
`alfalfa$alf␣eats␣`		`f␣eats␣alfalfa$a`**`l`**
`lfalfa$alf␣eats␣a`		`fa$alf␣eats␣alfa`**`l`**
`falfa$alf␣eats␣al`		`falfa$alf␣eats␣a`**`l`**
`alfa$alf␣eats␣alf`		`lf␣eats␣alfalfa$`**`a`**
`lfa$alf␣eats␣alfa`		`lfa$alf␣eats␣alf`**`a`**
`fa$alf␣eats␣alfal`		`lfalfa$alf␣eats␣`**`a`**
`a$alf␣eats␣alfalf`		`s␣alfalfa$alf␣ea`**`t`**
`$alf␣eats␣alfalfa`		`ts␣alfalfa$alf␣e`**`a`**

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

- more direct methods now also available

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

BWT – Properties

r	↓	$L[r]$
0	`$alf␣eats␣alfalf`**`a`**	16
1	`␣alfalfa$alf␣eat`**`s`**	8
2	`␣eats␣alfalfa$al`**`f`**	3
3	`a$alf␣eats␣alfal`**`f`**	15
4	`alf␣eats␣alfalfa`**`$`**	0
5	`alfa$alf␣eats␣al`**`f`**	12
6	`alfalfa$alf␣eats`**`␣`**	9
7	`ats␣alfalfa$alf␣`**`e`**	5
8	`eats␣alfalfa$alf`**`␣`**	4
9	`f␣eats␣alfalfa$a`**`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

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

```
have␣had␣hadnt␣hasnt␣havent␣has␣what$
ave␣had␣hadnt␣hasnt␣havent␣has␣what$h
ve␣had␣hadnt␣hasnt␣havent␣has␣what$ha
e␣had␣hadnt␣hasnt␣havent␣has␣what$hav
␣had␣hadnt␣hasnt␣havent␣has␣what$have
had␣hadnt␣hasnt␣havent␣has␣what$have␣
ad␣hadnt␣hasnt␣havent␣has␣what$have␣h
d␣hadnt␣hasnt␣havent␣has␣what$have␣ha
␣hadnt␣hasnt␣havent␣has␣what$have␣had
hadnt␣hasnt␣havent␣has␣what$have␣had␣
adnt␣hasnt␣havent␣has␣what$have␣had␣h
dnt␣hasnt␣havent␣has␣what$have␣had␣ha
nt␣hasnt␣havent␣has␣what$have␣had␣had
t␣hasnt␣havent␣has␣what$have␣had␣hadn
␣hasnt␣havent␣has␣what$have␣had␣hadnt
hasnt␣havent␣has␣what$have␣had␣hadnt␣
asnt␣havent␣has␣what$have␣had␣hadnt␣h
snt␣havent␣has␣what$have␣had␣hadnt␣ha
nt␣havent␣has␣what$have␣had␣hadnt␣has
t␣havent␣has␣what$have␣had␣hadnt␣hasn
␣havent␣has␣what$have␣had␣hadnt␣hasnt
havent␣has␣what$have␣had␣hadnt␣hasnt␣
avent␣has␣what$have␣had␣hadnt␣hasnt␣h
vent␣has␣what$have␣had␣hadnt␣hasnt␣ha
ent␣has␣what$have␣had␣hadnt␣hasnt␣hav
nt␣has␣what$have␣had␣hadnt␣hasnt␣have
t␣has␣what$have␣had␣hadnt␣hasnt␣haven
␣has␣what$have␣had␣hadnt␣hasnt␣havent
has␣what$have␣had␣hadnt␣hasnt␣havent␣
as␣what$have␣had␣hadnt␣hasnt␣havent␣h
s␣what$have␣had␣hadnt␣hasnt␣havent␣ha
␣what$have␣had␣hadnt␣hasnt␣havent␣has
what$have␣had␣hadnt␣hasnt␣havent␣has␣
hat$have␣had␣hadnt␣hasnt␣havent␣has␣w
at$have␣had␣hadnt␣hasnt␣havent␣has␣wh
t$have␣had␣hadnt␣hasnt␣havent␣has␣wha
$have␣had␣hadnt␣hasnt␣havent␣has␣what
```

```
$have␣had␣hadnt␣hasnt␣havent␣has␣wha t
␣had␣hadnt␣hasnt␣havent␣has␣what$hav e
␣hadnt␣hasnt␣havent␣has␣what$have␣ha d
␣has␣what$have␣had␣hadnt␣hasnt␣haven t
␣hasnt␣havent␣has␣what$have␣had␣hadn t
␣havent␣has␣what$have␣had␣hadnt␣hasn t
␣what$have␣had␣hadnt␣hasnt␣havent␣ha s
ad␣hadnt␣hasnt␣havent␣has␣what$have␣ h
adnt␣hasnt␣havent␣has␣what$have␣had␣ h
as␣what$have␣had␣hadnt␣hasnt␣havent␣ h
asnt␣havent␣has␣what$have␣had␣hadnt␣ h
at$have␣had␣hadnt␣hasnt␣havent␣has␣w h
ave␣had␣hadnt␣hasnt␣havent␣has␣what$ h
avent␣has␣what$have␣had␣hadnt␣hasnt␣ h
d␣hadnt␣hasnt␣havent␣has␣what$have␣h a
dnt␣hasnt␣havent␣has␣what$have␣had␣h a
e␣had␣hadnt␣hasnt␣havent␣has␣what$ha v
ent␣has␣what$have␣had␣hadnt␣hasnt␣ha v
had␣hadnt␣hasnt␣havent␣has␣what$have ␣
hadnt␣hasnt␣havent␣has␣what$have␣had ␣
has␣what$have␣had␣hadnt␣hasnt␣havent ␣
hasnt␣havent␣has␣what$have␣had␣hadnt ␣
hat$have␣had␣hadnt␣hasnt␣havent␣has␣ w
have␣had␣hadnt␣hasnt␣havent␣has␣what $
havent␣has␣what$have␣had␣hadnt␣hasnt ␣
nt␣has␣what$have␣had␣hadnt␣hasnt␣hav e
nt␣hasnt␣havent␣has␣what$have␣had␣ha d
nt␣havent␣has␣what$have␣had␣hadnt␣ha s
s␣what$have␣had␣hadnt␣hasnt␣havent␣h a
snt␣havent␣has␣what$have␣had␣hadnt␣h a
t$have␣had␣hadnt␣hasnt␣havent␣has␣wh a
t␣has␣what$have␣had␣hadnt␣hasnt␣have n
t␣hasnt␣havent␣has␣what$have␣had␣had n
t␣havent␣has␣what$have␣had␣hadnt␣has n
ve␣had␣hadnt␣hasnt␣havent␣has␣what$h a
vent␣has␣what$have␣had␣hadnt␣hasnt␣h a
what$have␣had␣hadnt␣hasnt␣havent␣has ␣
```

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

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

Run-length BWT Compression

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

Example:

$$S = \texttt{alf␣eats␣alfalfa\$}$$

$$B = \texttt{asff\$f␣e␣lllaaata}$$

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

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

Larger Example:

$$S = \texttt{have␣had␣hadnt␣hasnt␣havent␣has␣what\$}$$

$$B = \texttt{tedtttshhhhhhaavv␣␣␣␣w\$␣edsaaannnaa␣}$$

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

- 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?*

- **"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 $(\text{char}, \text{next entry})$

Example:

B = `ard$rcaaaabb`

S = `abracadabra$`

D

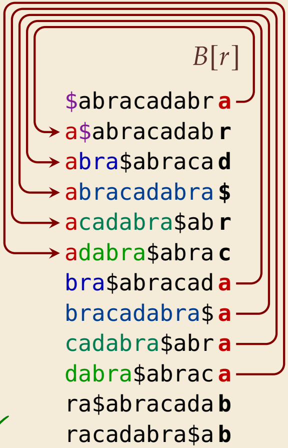
0	(a, 0)
1	(r, 1)
2	(d, 2)
3	($, 3)
4	(r, 4)
5	(c, 5)
6	(a, 6)
7	(a, 7)
8	(a, 8)
9	(a, 9)
10	(b, 10)
11	(b, 11)

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 – The magic revealed

- Inverse BWT very easy to compute:
  - only sort individual characters in B (not suffixes)
  - $\leadsto$ $O(n)$ with counting sort
- *but why does this work!?*
- decode char by char
  - can find unique \$ $\leadsto$ 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
  - $\leadsto$ 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
  - $\leadsto$ stably sorting $(B[r], r)$ by first entry enough ✓



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

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	$\mathrm{sort}(D)$
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 $\mathrm{sort}(T) = \mathrm{sort}(B)$
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.

#occurrences of c

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

	0	1	2	3	4	5	6	7	8
$T[0..9)$	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

$\text{select}_a(T, r)$	/	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

	0	1	2	3	4	5	6	7	8
$\text{sort}(T)$	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 $O[c] \le i < O[c+1]$
- 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,
- $\mathrm{rank}_c(T, i)$ and $\mathrm{select}_c(T, r)$ in $O(\log \sigma)$ time, and
- occupies $\sim n \lg \sigma$ *bits* of space. *(Further compression possible!)* ⇝ *Advanced Data Structures*
- The generalized $\sigma\text{-}\mathrm{rank}_c(T, i) = \mathrm{rank}_c(T, i) + \sum_{c' < c} |T|_{c'}$ is also supported in $O(\log \sigma)$ time

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

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

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

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!

⇝ 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.

$P = \texttt{ana}$

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

Backwards Search – Code

Recall total rank operation supported by wavelet trees

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

```
procedure backwardSearch($B[0..n]$, $P[0..m)$)
    // $B[0..n]$ given as wavelet tree
    // returns range $[s..e)$ of ranks for suffixes starting with $P$
    $c := P[m-1]$
    $s := \sigma\text{-rank}_c(B,0)$
    $e := \sigma\text{-rank}_c(B,n)$
    for $j := m-2, m-3, \ldots, 0$
        if $s \ge e$ break // no matches
        $c := P[j]$
        $s := \sigma\text{-rank}_c(B,s)$
        $e := \sigma\text{-rank}_c(B,e)$
    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* $r \in [s..e)$

⇝ 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?