Text Indexing

Lecture 08: FM-Index and *r*-Index

Florian Kurpicz

Recap: Wavelet Trees

$[0, 7]$

0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0

0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Recap: Wavelet Trees

$[0, 7]$

0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0

0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Recap: Wavelet Trees

[0, 7]

0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0

[0, 3]

0	1	1	2	3
0	0	0	1	1

[4, 7]

6	7	5	4	6
1	1	0	0	1

0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Recap: Wavelet Trees

[0, 7]

0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0

[0, 3]

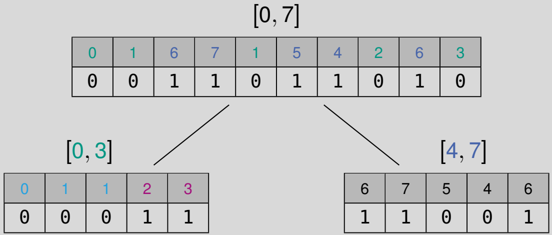
0	1	1	2	3
0	0	0	1	1

[4, 7]

6	7	5	4	6
1	1	0	0	1

0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Recap: Wavelet Trees



0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Recap: Wavelet Trees

[0, 7]

0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0

[0, 3]

0	1	1	2	3
0	0	0	1	1

[4, 7]

6	7	5	4	6
1	1	0	0	1

[0, 1]

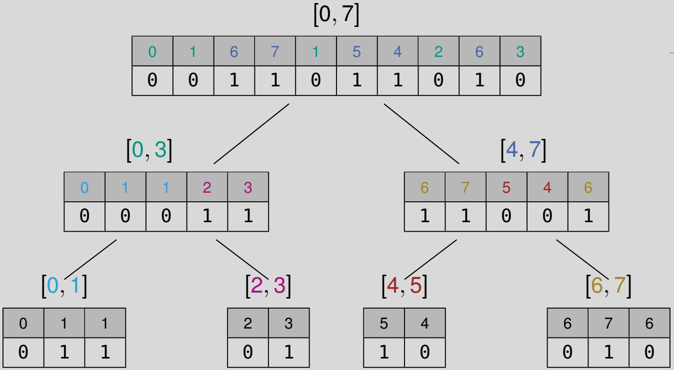
0	1	1
0	1	1

[2, 3]

2	3
0	1

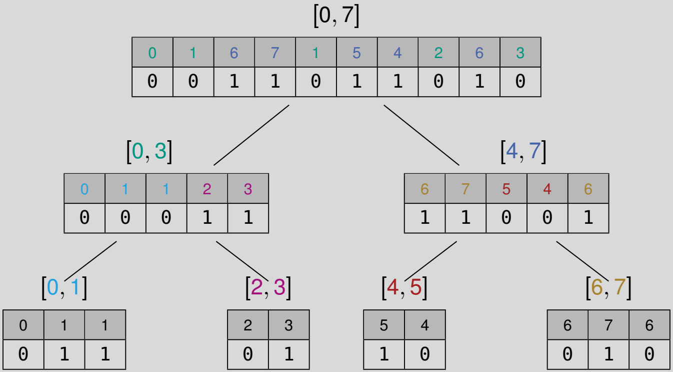
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Recap: Wavelet Trees



0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Recap: Wavelet Trees

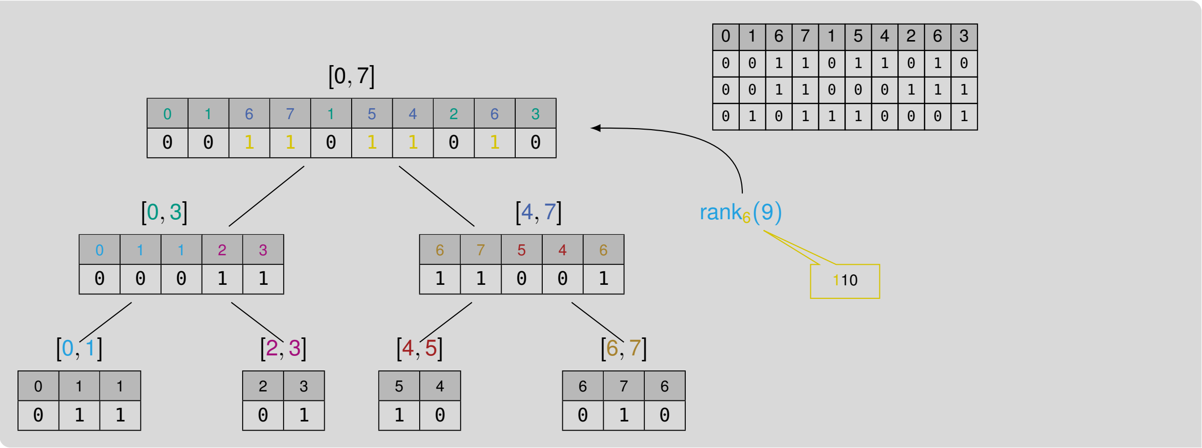


0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

$\mathrm{rank}_6(9)$

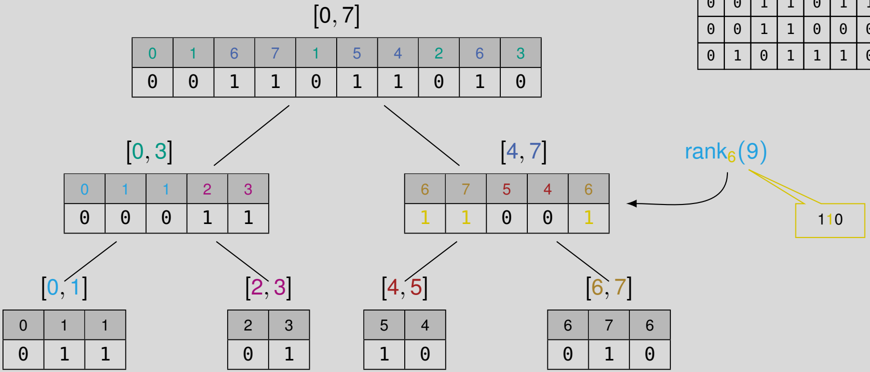
110

Recap: Wavelet Trees



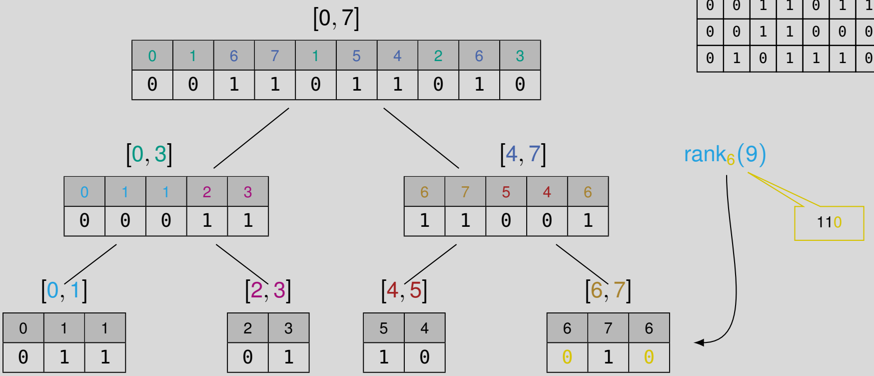
Recap: Wavelet Trees

0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

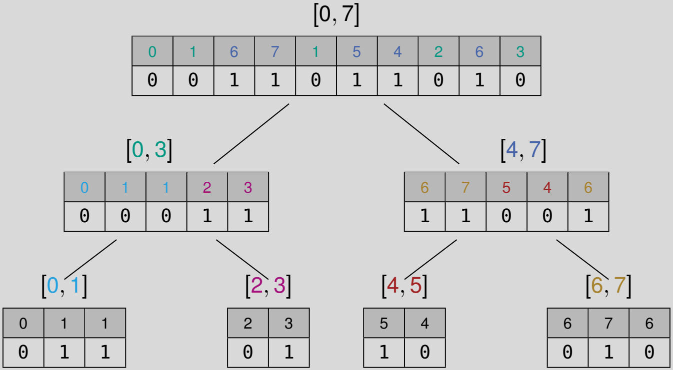


Recap: Wavelet Trees

0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

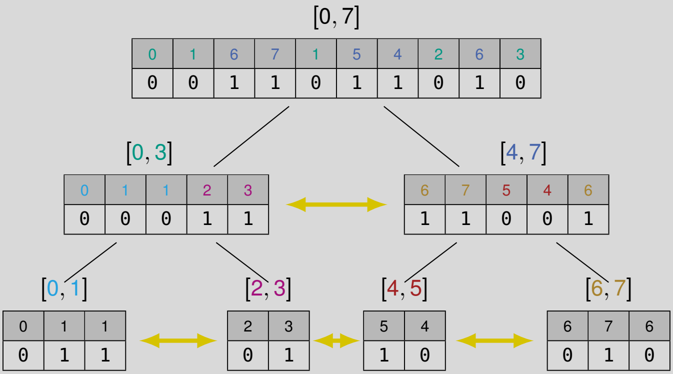


Recap: Wavelet Trees



0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Recap: Wavelet Trees



0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Recap: Compressed Wavelet Trees

0	1	3	7	1	5	4	2	6	3
0	1	1	0	1	0	0	0	0	1
0	7	5	4	2	6	1	3	1	3
0	1	0	0	0	1	1	0	1	0
0	5	4	2	7	6				
1	0	0	1	0	1				
5	4	0	2						
0	1	1	0						

- intervals are only missing to the right (white space)
- no holes allow for easy querying

- build wavelet tree for compressed text
- compress text using bit-wise negated canonical Huffman-codes

Recap: Compressed Wavelet Trees

0	1	3	7	1	5	4	2	6	3
0	1	1	0	1	0	0	0	0	1
0	7	5	4	2	6	1	3	1	3
0	1	0	0	0	1	1	0	1	0
0	5	4	2	7	6				
1	0	0	1	0	1				
5	4	0	2						
0	1	1	0						

- intervals are only missing to the right (white space)
- no holes allow for easy querying

- build wavelet tree for compressed text
- compress text using bit-wise negated canonical Huffman-codes

- can a wavelet tree be compressed further?

Bit Vector Compression (1/2)

- compress (sparse) bit vectors
- bit vector contains k one bits
- use $O(k \lg \frac{n}{k}) + o(n)$ bits
- retrieve $\Theta(\lg n)$ bits at the same time
- similar to *rank* data structure

Bit Vector Compression (1/2)

- compress (sparse) bit vectors
- bit vector contains k one bits
- use $O(k \lg \frac{n}{k}) + o(n)$ bits
- retrieve $\Theta(\lg n)$ bits at the same time
- similar to *rank* data structure

- split bit vector into (super-)blocks
- blocks of size $s = \frac{\lg n}{2}$
- super-blocks of size $s' = s^2$

Bit Vector Compression (1/2)

- compress (sparse) bit vectors
- bit vector contains k one bits
- use $O(k \lg \frac{n}{k}) + o(n)$ bits
- retrieve $\Theta(\lg n)$ bits at the same time
- similar to *rank* data structure

- split bit vector into (super-)blocks
- blocks of size $s = \frac{\lg n}{2}$
- super-blocks of size $s' = s^2$

Array C

- number of ones in i-th block

Bit Vector Compression (1/2)

- compress (sparse) bit vectors
- bit vector contains k one bits
- use $O(k \lg \frac{n}{k}) + o(n)$ bits
- retrieve $\Theta(\lg n)$ bits at the same time
- similar to *rank* data structure

- split bit vector into (super-)blocks
- blocks of size $s = \frac{\lg n}{2}$
- super-blocks of size $s' = s^2$

Array C

- number of ones in i-th block

Lookup-Tables L_i

- for $i \in [0, s]$ store lookup-table containing all bit vectors with i one bits

Bit Vector Compression (1/2)

- compress (sparse) bit vectors
- bit vector contains k one bits
- use $O(k \lg \frac{n}{k}) + o(n)$ bits
- retrieve $\Theta(\lg n)$ bits at the same time
- similar to *rank* data structure

- split bit vector into (super-)blocks
- blocks of size $s = \frac{\lg n}{2}$
- super-blocks of size $s' = s^2$

Array C

- number of ones in i-th block

Lookup-Tables L_i

- for $i \in [0, s]$ store lookup-table containing all bit vectors with i one bits

- use variable-length codes to identify content of block
- concatenate all codes in bit vector V

Bit Vector Compression (1/2)

- compress (sparse) bit vectors
- bit vector contains k one bits
- use $O(k \lg \frac{n}{k}) + o(n)$ bits
- retrieve $\Theta(\lg n)$ bits at the same time
- similar to *rank* data structure

- split bit vector into (super-)blocks
- blocks of size $s = \frac{\lg n}{2}$
- super-blocks of size $s' = s^2$

Array C

- number of ones in i-th block

Lookup-Tables L_i

- for $i \in [0, s]$ store lookup-table containing all bit vectors with i one bits

- use variable-length codes to identify content of block
- concatenate all codes in bit vector V

Bit Vector V

- let k_i be number of ones in i-th block
- use $\lceil \lg \binom{s}{k_i} \rceil$ bits to encode block ⓘ position in lookup-table
- concatenate all codes

Bit Vector Compression (2/2)

Array *SBlock*

- for every super-block i, $SBlock[i]$ contains position of encoding of first block in i-th super-block in V
- $\lceil \lg n \rceil$ bits per entry

Bit Vector Compression (2/2)

Array *SBlock*

- for every super-block i, $SBlock[i]$ contains position of encoding of first block in i-th super-block in V
- $\lceil \lg n \rceil$ bits per entry

Array *Block*

- for every block i, $Block[i]$ contains position of encoding of i-th block in V relative to its super-block
- $O(\lg \lg n)$ bits per entry

Bit Vector Compression (2/2)

Array *SBlock*

- for every super-block i, $\mathit{SBlock}[i]$ contains position of encoding of first block in i-th super-block in V
- $\lceil \lg n \rceil$ bits per entry

Array *Block*

- for every block i, $\mathit{Block}[i]$ contains position of encoding of i-th block in V relative to its super-block
- $O(\lg \lg n)$ bits per entry

Lemma: Compressed Bit Vectors

A bit vector of size n containing k ones can be represented using $O(k \lg \frac{n}{k}) + o(n)$ bits allowing $O(1)$ time access to individual bits

Bit Vector Compression (2/2)

Array *SBlock*

- for every super-block i, $SBlock[i]$ contains position of encoding of first block in i-th super-block in V
- $\lceil \lg n \rceil$ bits per entry

Array *Block*

- for every block i, $Block[i]$ contains position of encoding of i-th block in V relative to its super-block
- $O(\lg \lg n)$ bits per entry

Lemma: Compressed Bit Vectors

A bit vector of size n containing k ones can be represented using $O(k \lg \frac{n}{k}) + o(n)$ bits allowing $O(1)$ time access to individual bits

Proof (Sketch space requirements)

- $|C| = O(\frac{n}{s} \lg s) = o(n)$ bits
- $|SBlock| = O(\frac{n}{s'} \lg n) = o(n)$ bits
- $|Block| = O(\frac{n}{s} \lg s) = o(n)$ bits
- $\sum_{k=0}^{s} |L_k| \leq (s+1)2^s s = o(n)$ bits
- $|V| = \sum_{i=1}^{\lceil n/s \rceil} \lceil \lg \binom{s}{k_i} \rceil \leq \lg \binom{n}{k} + n/s \leq \lg((n/k)^k) + n/s = k \lg \frac{n}{k} + O(\frac{n}{\lg n})$ bits

Recap: Backwards Search in the BWT

Karlsruhe Institute of Technology

Function *BackwardsSearch*$(P[1..n], C, rank)$**:**

- $s = 1, e = n$
- **for** $i = m, \ldots, 1$ **do**
  - $s = C[P[i]] + rank_{P[i]}(s - 1) + 1$
  - $e = C[P[i]] + rank_{P[i]}(e)$
  - **if** $s > e$ **then**
    - **return** $\emptyset$
- **return** $[s, e]$

- no access to text or *SA* required
- no binary search
- existential queries are easy
- counting queries are easy
- reporting queries require additional information
- example on the board

The FM-Index [FM00]

Building Blocks of FM-Index

- wavelet tree on BWT providing *rank*-function
- C-array
- sampled suffix array with sample rate s
- bit vector marking sampled suffix array positions

Lemma: FM-Index

Given a text T of length n over an alphabet of size σ, the FM-index requires $O(n \lg \sigma)$ bits of space and can answer counting queries in $O(m \lg \sigma)$ time and reporting queries in $O(occ + m \lg \sigma)$ time

The FM-Index [FM00]

Building Blocks of FM-Index

- wavelet tree on BWT providing *rank*-function
- C-array
- sampled suffix array with sample rate s
- bit vector marking sampled suffix array positions

Lemma: FM-Index

Given a text T of length n over an alphabet of size σ, the FM-index requires $O(n \lg \sigma)$ bits of space and can answer counting queries in $O(m \lg \sigma)$ time and reporting queries in $O(occ + m \lg \sigma)$ time

Space Requirements

- wavelet tree: $n\lceil \lg \sigma \rceil (1 + o(1))$ bits
- C-array: $\sigma \lceil \lg n \rceil$ bits ℹ $n(1 + o(1))$ bits if $\sigma \geq \frac{n}{\lg n}$
- sampled suffix array: $\frac{n}{s} \lceil \lg n \rceil$ bits
- bit vector: $n(1 + o(1))$ bits

- space and time bounds can be achieved with $s = \lg_\sigma n$

Conclusion FM-Index

- FM-index is easy to compress
- wavelet tree on *BWT* can be compressed
- bit vector can be compressed

- very small in comparison with suffix tree or suffix array
- compression does not make use of structure of *BWT* ℹ wavelet trees are compressed using Huffman-codes

Conclusion FM-Index

- FM-index is easy to compress
- wavelet tree on *BWT* can be compressed
- bit vector can be compressed

- very small in comparison with suffix tree or suffix array
- compression does not make use of structure of *BWT* ℹ wavelet trees are compressed using Huffman-codes

Definition: Run (simplified, recap)

Given a text T of length n, we call its substring $T[i..j]$ a **run**, if

- $T[k] = T[\ell]$ for all $k, \ell \in [i, j]$ and
- $T[i-1] \neq T[i]$ and $T[j+1] \neq T[j]$

ℹ (To be more precise, this is a definition for a run of a periodic substring with smallest period 1, but this is not important for this lecture)

	1	2	3	4	5	6	7	8	9	0	11	12	13
L	a	b	$	c	c	b	b	a	a	a	a	b	b

Measures of Repetitiveness (Excerpt)

$b < z < g < r < e$

b — # phrases in bidirectional macro scheme

a	b	r	a	c	a	d	a	b	r	a

(6,2)r(11,1)cad(1,4)

NP-hard

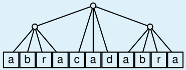
z — # phrases in Lempel-Ziv parse

a	b	r	a	c	a	d	a	b	r	a

abr(1,1)c(1,1)d(1,4)

$O(n)$ time

g — size of smallest context free grammar



NP-hard*

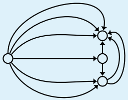
r — # of runs in Burrows-Wheeler transform

a	r	d	$	r	c	a	a	a	a	b	b

(1,a)(1,r)(1,d)(1,$)(1,r)(1,c)(4,a)(2,b)

$O(n)$ time

e — # of nodes and edges in CDAWG



$O(n)$ time

* there are good heuristics

Measures of Repetitiveness (Excerpt)

order of magnitude in practice

$b < z < g < r < e$

b: # phrases in bidirectional macro scheme

a	b	r	a	c	a	d	a	b	r	a

(6,2)r(11,1)cad(1,4)

NP-hard

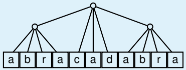
z: # phrases in Lempel-Ziv parse

a	b	r	a	c	a	d	a	b	r	a

abr(1,1)c(1,1)d(1,4)

$O(n)$ time

g: size of smallest context free grammar



NP-hard*

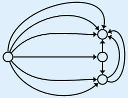
r: # of runs in Burrows-Wheeler transform

a	r	d	$	r	c	a	a	a	a	b	b

(1,a)(1,r)(1,d)(1,$)(1,r)(1,c)(4,a)(2,b)

$O(n)$ time

e: # of nodes and edges in CDAWG



$O(n)$ time

* there are good heuristics

Motivation: *r*-Index

Karlsruhe Institute of Technology

Measure for Compressibility

- k-th order empirical entropy H_k
- number of LZ factors z
- number of *BWT* runs r

Motivation: r-Index

Karlsruhe Institute of Technology

Measure for Compressibility

- k-th order empirical entropy H_k
- number of LZ factors z
- number of *BWT* runs r

- z and r not blind to repetitions
- how do they relate?

Motivation: r-Index

Measure for Compressibility

- k-th order empirical entropy H_k
- number of LZ factors z
- number of *BWT* runs r

- z and r not blind to repetitions
- how do they relate?

Lemma: *BWT* runs and LZ factors [KK20]

Given a text T of length n. Let z be the number of LZ77 factors and r the number of runs in T's *BWT*, then

$$r \in O(z \lg^2 n)$$

Motivation: r-Index

Measure for Compressibility

- k-th order empirical entropy H_k
- number of LZ factors z
- number of *BWT* runs r

- z and r not blind to repetitions
- how do they relate?

Lemma: *BWT* runs and LZ factors [KK20]

Given a text T of length n. Let z be the number of LZ77 factors and r the number of runs in T's *BWT*, then

$$r \in O(z \lg^2 n)$$

- more details in next lecture

Main Part of Backwards-Search

Function *BackwardsSearch*($P[1..n]$, C, *rank*):

```
s = 1, e = n
for i = m, ..., 1 do
    s = C[P[i]] + rank_{P[i]}(s − 1) + 1
    e = C[P[i]] + rank_{P[i]}(e)
    if s > e then
        return ∅
return [s, e]
```

Goals

- simulate *BWT* and *rank* on *BWT* in
- $O(r \lg n)$ bits of space

The r-Index [GNP20] (1/3)

Given a text T of length n over an alphabet Σ and its *BWT*, the r-index of this text consists of the following data structures

The r-Index [GNP20] (1/3)

Given a text T of length n over an alphabet Σ and its BWT, the r-index of this text consists of the following data structures

Array I

- $I[i]$ stores position of i-th run in BWT

The r-Index [GNP20] (1/3)

Given a text T of length n over an alphabet Σ and its BWT, the r-index of this text consists of the following data structures

Array I

- $I[i]$ stores position of i-th run in BWT

Array L'

- $L'[i]$ stores character of i-th run in BWT
- build wavelet tree for L'

The r-Index [GNP20] (1/3)

Given a text T of length n over an alphabet Σ and its BWT, the r-index of this text consists of the following data structures

Array I

- $I[i]$ stores position of i-th run in BWT

Array L'

- $L'[i]$ stores character of i-th run in BWT
- build wavelet tree for L'

Array R

- lengths of BWT runs stably sorted by runs' characters
- accumulate for each character by performing exclusive prefix sum over run lengths'

The r-Index [GNP20] (1/3)

Given a text T of length n over an alphabet Σ and its BWT, the r-index of this text consists of the following data structures

Array I

- $I[i]$ stores position of i-th run in BWT

Array L'

- $L'[i]$ stores character of i-th run in BWT
- build wavelet tree for L'

Array R

- lengths of BWT runs stably sorted by runs' characters
- accumulate for each character by performing exclusive prefix sum over run lengths'

Array C'

- $C'[\alpha]$ stores the start of the run lengths in R for each character $\alpha \in \Sigma$ starting at 0

The r-Index [GNP20] (1/3)

Given a text T of length n over an alphabet Σ and its BWT, the r-index of this text consists of the following data structures

Array I

- $I[i]$ stores position of i-th run in BWT

Array L'

- $L'[i]$ stores character of i-th run in BWT
- build wavelet tree for L'

Array R

- lengths of BWT runs stably sorted by runs' characters
- accumulate for each character by performing exclusive prefix sum over run lengths'

Array C'

- $C'[\alpha]$ stores the start of the run lengths in R for each character $\alpha \in \Sigma$ starting at 0

Bit Vector B

- compressed bit vector of length n containing ones at positions where BWT runs start and rank-support

The r-Index (2/3)

$rank_\alpha(BWT, i)$ with r-Index

- compute number j of run ($j = rank_1(B, i)$)
- compute position k in R ($k = C'[\alpha]$)
- compute number ℓ of α runs before the j-th run ($\ell = rank_\alpha(L', j-1)$)
- compute number k of αs before the j-th run ($k = R[k+\ell]$)
- compute character β of run ($\beta = L'[j]$)
- if $\alpha \neq \beta$ return k ℹ i is not in the run
- else return $k + i - I[j] + 1$ ℹ i is in the run

The r-Index (3/3)

Lemma: Space Requirements r-Index

Given a text T of length n over an alphabet of size σ that has r *BWT* runs, then its r-index requires

$$O(r \lg n) \text{ bits}$$

and can answer *rank*-queries on the *BWT* in $O(\lg \sigma)$.
Given a pattern of length m, the r-index can answer pattern matching queries in time

$$O(m \lg \sigma)$$

The r-Index (3/3)

Karlsruhe Institute of Technology

Lemma: Space Requirements r-Index

Given a text T of length n over an alphabet of size σ that has r *BWT* runs, then its r-index requires

$$O(r \lg n) \text{ bits}$$

and can answer *rank*-queries on the *BWT* in $O(\lg \sigma)$. Given a pattern of length m, the r-index can answer pattern matching queries in time

$$O(m \lg \sigma)$$

- what about reporting queries?

Locating Occurrences (Sketch)

Karlsruhe Institute of Technology

- modify backwards-search that it maintains $SA[e]$
- after backwards-search output $SA[e], SA[e-1], \ldots, SA[s]$
- in $O(r \lg n)$ bits and $O(occ \cdot \lg\lg r)$ time

Maintaining $SA[e]$

- sample SA positions at ends of runs
- if next character is $BWT[e]$, then next $SA[e']$ is $SA[e] - 1$
- otherwise locate end of run and extract sample

Output Result

- following LF not possible unbounded
- deduce $SA[i-1]$ from $SA[i]$
- character in L and F in same order
- only beginning of runs complicated
- for every character build predecessor data structure over sampled SA-values at end of runs
- associate with $\langle i, SA[i] \rangle$

The Move Data Structure [NT21]

Definition: Disjoint Interval Sequence

Let $I = (p_1, q_1), (p_2, q_2), \ldots, (p_k, q_k)$ be a sequence of k pairs of integers. We introduce a permutation π of $[1, k]$ and sequence $d_1, d_2, \ldots, d_k$ for I. π satisfies $q_{\pi[1]} \leq q_{\pi[2]} \leq \cdots \leq q_{\pi[k]}$, and $d_i = p_{i+1} - p_i$ for $i \in [1, k]$, where $p_{k+1} = n + 1$. We call the sequence I a disjoint interval sequence if it satisfies the following three conditions:

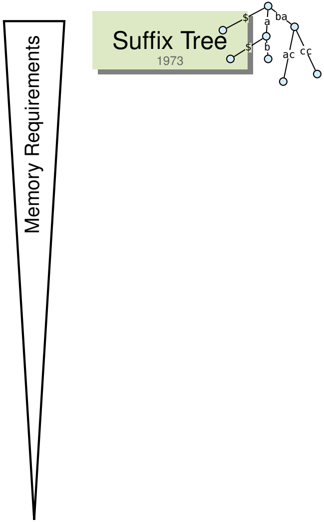
- $p_1 = 1 < p_2 < \cdots < p_k \leq n$
- $q_{\pi[1]} = 1$,
- $q_{\pi[i]} = q_{\pi[i-1]} + d_{\pi[i-1]}$ for each $i \in [2, k]$.

Move Query

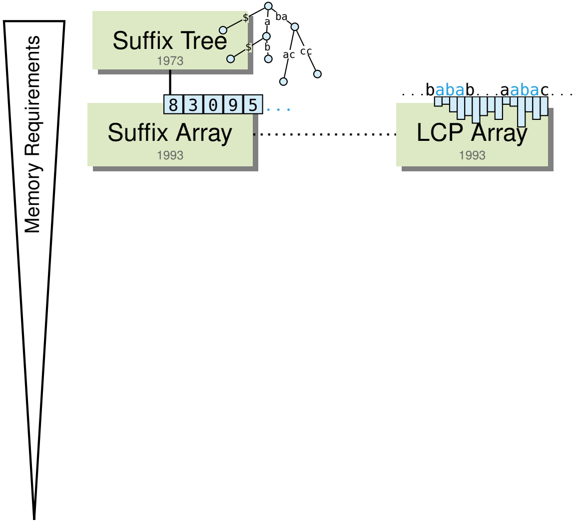
$$\mathrm{move}(i, x) = (i', x')$$

- i position in input interval
- x input interval
- i' position in output interval
- x' input interval covering i'

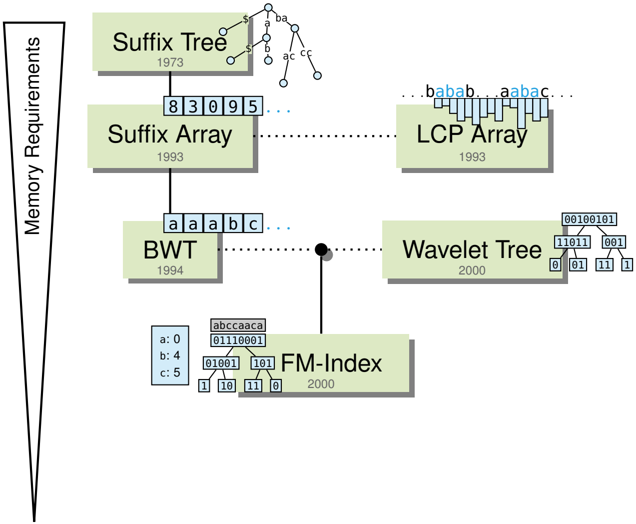
From the Suffix Tree to the *r*-Index—Questions?



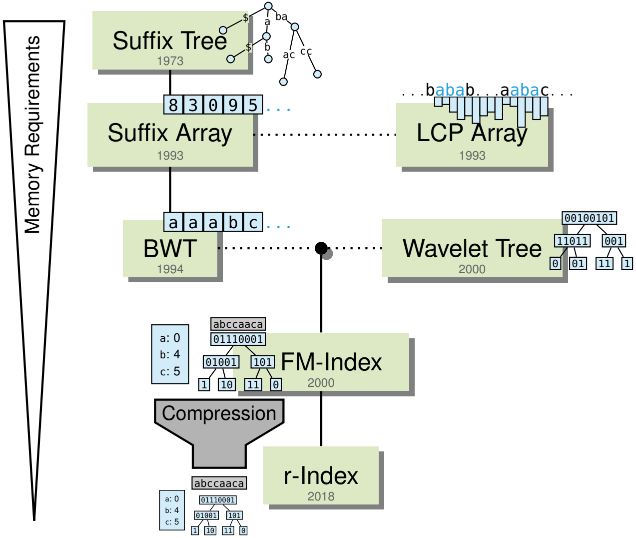
From the Suffix Tree to the *r*-Index—Questions?



From the Suffix Tree to the *r*-Index—Questions?



From the Suffix Tree to the *r*-Index—Questions?



Bibliography I

[FM00] Paolo Ferragina and Giovanni Manzini. "Opportunistic Data Structures with Applications". In: *FOCS*. IEEE Computer Society, 2000, pages 390–398. DOI: 10.1109/SFCS.2000.892127.

[GNP20] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. "Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space". In: *J. ACM* 67.1 (2020), 2:1–2:54. DOI: 10.1145/3375890.

[KK20] Dominik Kempa and Tomasz Kociumaka. "Resolution of the Burrows-Wheeler Transform Conjecture". In: *FOCS*. IEEE, 2020, pages 1002–1013. DOI: 10.1109/FOCS46700.2020.00097.

[NT21] Takaaki Nishimoto and Yasuo Tabei. "Optimal-Time Queries on BWT-Runs Compressed Indexes". In: *ICALP*. Volume 198. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 101:1–101:15. DOI: 10.4230/LIPIcs.ICALP.2021.101.