Text Indexing

Lecture 08: FM-Index and r-Index

Florian Kurpicz
Recap: Wavelet Trees

\[[0, 7]\]

\[
\begin{array}{cccccccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]
Recap: Wavelet Trees

$$[0, 7]$$

$$\begin{array}{cccccccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array}$$

$$\begin{array}{cccccccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
\end{array}$$
Recap: Wavelet Trees

[0, 7]

\[
\begin{array}{cccccccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]

[0, 3]

\[
\begin{array}{cccc}
0 & 1 & 1 & 2 & 3 \\
0 & 0 & 1 & 1 \\
\end{array}
\]

[4, 7]

\[
\begin{array}{cccc}
6 & 7 & 5 & 4 & 6 \\
1 & 1 & 0 & 0 & 1 \\
\end{array}
\]

→

\[
\begin{array}{cccccccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]
Recap: Wavelet Trees

[0, 7]

\[
\begin{array}{cccccccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]

[0, 3]

\[
\begin{array}{cccc}
0 & 1 & 1 & 2 & 3 \\
0 & 0 & 0 & 1 & 1 \\
\end{array}
\]

[4, 7]

\[
\begin{array}{cccc}
6 & 7 & 5 & 4 & 6 \\
1 & 1 & 0 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]
Recap: Wavelet Trees

![Wavelet Tree Diagram]

- **[0, 7]**
  - Subtrees: [0, 3], [4, 7]
  - Values: 0 1 6 7 1 5 4 2 6 3

- **[0, 3]**
  - Subtrees: 0 1 1 2 3
  - Values: 0 0 0 1 1

- **[4, 7]**
  - Subtrees: 0 1 6 7 5 4 6
  - Values: 1 1 0 0 1

- **[0, 10]**
  - Subtrees: 0 0 1 1 0 1 1 0 1 0
  - Values: 0 0 1 1 0 0 1 1 1

- **[0, 13]**
  - Subtrees: 0 1 0 1 1 1 0 0 0 1

- **[0, 16]**
  - Subtrees: 0 1 6 7 1 5 4 2 6 3

- **[0, 19]**
  - Subtrees: 0 0 1 1 0 1 1 0 1 0

- **[0, 22]**
  - Subtrees: 0 0 1 1 0 0 1 1 1

- **[0, 25]**
  - Subtrees: 0 1 0 1 1 1 0 0 0 1
Recap: Wavelet Trees
Recap: Wavelet Trees

Wavelet Trees are data structures used for efficient string and substring search. They are built from Wavelet Matrices, which are used to store and query bit vectors efficiently.

The Wavelet Tree for the range [0, 7] is constructed as follows:

- The root node represents the entire range [0, 7].
- The tree is split into two subtrees: [0, 3] and [4, 7].
- Each subtree is further split into two more subtrees until the leaf nodes are reached.
- The leaf nodes represent individual characters in the range.

The Wavelet Trees for the ranges [0, 1], [2, 3], [4, 5], and [6, 7] are shown with their corresponding bit vectors.

The bit vectors are used to efficiently query the presence of characters within the given ranges.

```
[0, 7]

[0, 3]

[4, 7]

[0, 1]

[2, 3]

[4, 5]

[6, 7]
```
Recap: Wavelet Trees
Recap: Wavelet Trees
Recap: Wavelet Trees

Wavelet Trees are data structures used for efficient rank and select queries on a string. They are built on top of Rank and Select Trees and are particularly useful for text indexing. The diagram shows a wavelet tree for the string "011101101010011100001". Each level of the tree corresponds to a substring of the input, and the leaves represent individual characters.

The tree is built level by level, with each level corresponding to a character in the string. The bottom level of the tree contains the characters, while the root contains the entire string. The intermediate levels store information about the characters in the substrings that they represent.

The rank and select operations are performed by traversing the tree from the root to the leaves. The rank operation counts the number of occurrences of a character up to a given position, while the select operation finds the position of a character in the string.
Recap: Wavelet Trees
Recap: Wavelet Trees
Recap: Wavelet Trees

[0, 7]

[0, 3]

[0, 1]

[2, 3]

[4, 5]

[6, 7]

[4, 7]
Recap: Compressed Wavelet Trees

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

- intervals are only missing to the right (white space)
- no holes allow for easy querying
Recap: Compressed Wavelet Trees

- Build wavelet tree for compressed text
- Compress text using bit-wise negated canonical Huffman-codes
- Can a wavelet tree be compressed further?

- Intervals are only missing to the right (white space)
- No holes allow for easy querying
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
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 $s' = s^2$
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
compress (sparse) bit vectors
- bit vector contains $k$ one bits
- use $O(k \log \frac{n}{k}) + o(n)$ 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

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

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

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 \( V \)

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 \( O(k \log \frac{n}{k}) + o(n) \) bits
- retrieve \( \Theta(\log n) \) bits at the same time
- similar to rank data structure

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

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 \( V \)

Array \( C \)

- number of ones in \( i \)-th block

Bit Vector \( V \)

- let \( k_i \) be number of ones in \( i \)-th block
- use \( \lceil \log \binom{s}{k_i} \rceil \) bits to encode block \( i \) 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
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
Array \textit{SBlock}

- for every super-block \(i\), \(\textit{SBlock}[i]\) contains position of encoding of first block in \(i\)-th super-block in \(V\)
- \([\lg n]\) bits per entry

Array \textit{Block}

- for every block \(i\), \(\textit{Block}[i]\) 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 \( V \)
- \( \lceil \log n \rceil \) 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
- \( O(\log \log n) \) bits per entry

---

#### Lemma: Compressed Bit Vectors
A bit vector of size \( n \) containing \( k \) ones can be represented using \( O(k \log \frac{n}{k}) + o(n) \) bits allowing \( O(1) \) time access to individual bits

#### Proof (Sketch space requirements)
- \( |C| = O(\frac{n}{s} \log s) = o(n) \) bits
- \( |SBlock| = O(\frac{n}{s} \log n) = o(n) \) bits
- \( |Block| = O(\frac{n}{s} \log s) = o(n) \) bits
- \( \sum_{k=0}^{s} |L_k| \leq (s + 1)2^s s = o(n) \) bits
- \( |V| = \sum_{i=1}^{[n/s]} \lceil \log \left( \frac{s}{k_i} \right) \rceil \leq \log \left( \frac{n}{k} \right) + \frac{n}{s} \leq \log((n/k)^k) + \frac{n}{s} = k \log \frac{n}{k} + O(\frac{n}{\log n}) \) bits
Recap: Backwards Search in the BWT

Function `BackwardsSearch(P[1..n], C, rank)`:  
1. \( s = 1, e = n \)
2. \textbf{for} \( i = m, \ldots, 1 \) \textbf{do}
3. \hspace{1em} \( s = C[P[i]] + \text{rank}_{P[i]}(s - 1) + 1 \)
4. \hspace{1em} \( e = C[P[i]] + \text{rank}_{P[i]}(e) \)
5. \hspace{1em} \textbf{if} \( s > e \) \textbf{then}
6. \hspace{2em} \textbf{return} \( \emptyset \)
7. \textbf{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 $\sigma$, 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 $O(\text{occ} + m \lg \sigma)$ time
## The FM-Index \[FM00\]

### Building Blocks of FM-Index
- wavelet tree on BWT providing \textit{rank}-function
- \textit{C}-array
- sampled suffix array with sample rate \(s\)
- bit vector marking sampled suffix array positions

### Space Requirements
- wavelet tree: \(n\lg \sigma)(1 + o(1))\) bits
- \textit{C}-array: \(\sigma \lg n\) bits \(\Theta n(1 + o(1))\) bits if \(\sigma \geq \frac{n}{\lg n}\)
- sampled suffix array: \(\frac{n}{s}\lg n\) bits
- bit vector: \(n(1 + o(1))\) bits

### Lemma: FM-Index
Given a text \(T\) of length \(n\) over an alphabet of size \(\sigma\), 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 \(O(\text{occ} + m \lg \sigma)\) time

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 $BWT$ 
- 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
- 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 $T[j + 1] \neq T[j]$

(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.)
Measures of Repetitiveness (Excerpt)

- Phrases in bidirectional macro scheme: $b$
- Lempel-Ziv parsed phrases: $z$
- Size of smallest context-free grammar: $g$
- #. of runs in Burrows-Wheeler transform: $r$
- #. of nodes and edges in CDAWG: $e$

For the given sequences:

- $abracadabra$: NP-hard
- $abracadabra$: $O(n)$ time
- $abracadabra$: NP-hard*
- $abracadabra$: $O(n)$ time
- $abracadabra$: $O(n)$ time

* there are good heuristics
Measures of Repetitiveness (Excerpt)

- b: # of phrases in bidirectional macro scheme
- z: # of phrases in Lempel-Ziv parse tree
- g: size of smallest context-free grammar
- r: # of runs in Burrows-Wheeler transform
- e: # of edges in CDAWG

- abracadabra
  (6,2)r(1,1)c(1,4)
- abracadabra
  abr(1,1)c(1,1)d(1,4)
- abracadabra
  abr(1,1)c(1,1)d(1,4)
- abracadabra
  abr(1,1)c(1,1)d(1,4)

NP-hard  O(n) time  NP-hard*  O(n) time  O(n) time

* there are good heuristics
Motivation: \textit{r-Index}

Measure for Compressibility

- \( k \)-th order empirical entropy \( H_k \)
- number of LZ factors \( z \)
- number of \( BWT \) runs \( r \)
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?
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

\[
r \in O(z \log^2 n)
\]
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

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

- more details in next lecture
Main Part of Backwards-Search

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

1. $s = 1, e = n$
2. for $i = m, \ldots, 1$ do
   3. $s = C[P[i]] + rank_{P[i]}(s - 1) + 1$
   4. $e = C[P[i]] + rank_{P[i]}(e)$
   5. if $s > e$ then
      6. return $\emptyset$
   7. return $[s, e]$

**Goals**

- simulate BWT and rank on BWT in $O(r \lg n)$ bits of space
Given a text $T$ of length $n$ over an alphabet $\Sigma$ 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 $\Sigma$ 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 $BWT$
Given a text $T$ of length $n$ over an alphabet $\Sigma$ 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 \cite{GNP20} (1/3)

Given a text $T$ of length $n$ over an alphabet $\Sigma$ 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 \[\text{[GNP20]}\] (1/3)

Given a text $T$ of length $n$ over an alphabet $\Sigma$ 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
Given a text $T$ of length $n$ over an alphabet $\Sigma$ 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)

$\text{rank}_\alpha(BWT, i)$ with $r$-Index

- Compute number $j$ of run ($j = \text{rank}_1(B, i)$)
- Compute position $k$ in $R$ ($k = C'[\alpha]$)
- Compute number $\ell$ of $\alpha$ runs before the $j$-th run
  ($\ell = \text{rank}_\alpha(L', j - 1)$)
- Compute number $k$ of $\alpha$s before the $j$-th run
  ($k = R[k + \ell]$)
- Compute character $\beta$ of run ($\beta = L'[j]$)
- If $\alpha \neq \beta$ return $k \uparrow i$ is not in the run
- Else return $k + i - l[j] + 1 \uparrow i$ is in the run
Lemma: Space Requirements $r$-Index

Given a text $T$ of length $n$ over an alphabet of size $\sigma$ 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)

**Lemma: Space Requirements $r$-Index**

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

$$O(r \lg n)$$ 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)

- 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 \( \uparrow \) 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 \( \pi \) of \([1, k]\) and sequence \( d_1, d_2, \ldots, d_k \) for \( I \). \( \pi \) 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**

\[
\text{move}(i, x) = (i', x')
\]

- \( i \) 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?

- Suffix Tree (1973)
- Suffix Array (1993)
- LCP Array (1993)

Memory Requirements
From the Suffix Tree to the $r$-Index—Questions?

- Suffix Tree: 1973
- Suffix Array: 1993
- LCP Array: 1993
- BWT: 1994
- Wavelet Tree: 2000
- FM-Index: 2000
- $r$-Index: 2018

Memory Requirements

**FM-Index**
- $a$: 0
- $b$: 4
- $c$: 5

**Suffix Array**
- 8 3 0 9 5...

**LCP Array**
- ...babab aabac...

**Wavelet Tree**
- 0 01 001 11011 00100101

**Suffix Tree**
- ...a a a b c...

**BWT**
- abccaaca: 01110001 01001 101 110 1101 01110001

**Compression**
- abccaaca: 01110001 01001 101 110 1101 01110001

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

