Recap: Wavelet Trees

\[ [0, 7] \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>5</th>
<th>4</th>
<th>2</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

→

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>5</th>
<th>4</th>
<th>2</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
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 1 6 7 1 5 4 2 6 3
0 0 1 1 0 1 1 0 1 0

0 1 1 0 0 1 1 1
0 1 1 1 1 0 0 1

0 1 0 1 1 1 0 0 0 1

[0, 7]

[0, 3]

[4, 7]
Recap: Wavelet Trees

Wavelet Trees are data structures that allow for efficient rank and select operations on bit vectors. They are often used in text indexing and compression algorithms. The diagram shows a Wavelet Tree for the interval [0, 7], which is divided into sub-intervals [0, 3] and [4, 7]. The binary representation of each interval is shown in the table on the right.
Recap: Wavelet Trees

\[ [0, 7] \]

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

\[ [0, 3] \]

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

\[ [4, 7] \]

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

Wavelet Trees are data structures used in string processing and bioinformatics for efficient query on strings. They are constructed to support various operations such as rank and select queries on the strings.

In the diagram, we see a wavelet tree structure with leaves labeled [0, 1], [0, 3], [0, 7], [2, 3], and [4, 7]. Each node in the tree represents a segment of the string, and the leaves correspond to specific ranges of the string.

The tree is built by partitioning the string into smaller segments, each of which is represented by a node in the tree. The leaves of the tree correspond to the smallest segments, and the root node represents the entire string.

The wavelet tree is useful for solving problems such as finding the kth smallest character in a given range of the string or finding the longest substring that satisfies a given property.
Recap: Wavelet Trees

![Wavelet Tree Diagram]

<table>
<thead>
<tr>
<th>Interval</th>
<th>Node</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0, 1]</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[2, 3]</td>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4, 5]</td>
<td></td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6, 7]</td>
<td></td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interval</th>
<th>Node</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0, 3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4, 7]</td>
<td></td>
<td>6</td>
<td>7</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interval</th>
<th>Node</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0, 7]</td>
<td></td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interval</th>
<th>Node</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0, 1]</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[2, 3]</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>[4, 5]</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[6, 7]</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Recap: Wavelet Trees

Wavelet trees are a data structure used for various applications, including text indexing. They store information about the presence of elements in a set, allowing for efficient searching and querying operations. The diagram illustrates how wavelet trees are constructed and how they can be used to represent intervals and their properties.
Recap: Wavelet Trees

```
[0, 7]

[0, 3]

[0, 1]

[2, 3]

[4, 5]

[6, 7]

[0, 3, 1]

[2, 3, 3]

[4, 5, 3]

[6, 7, 3]

0 1 1 0 0 1 1 1

0 0 0 1 1 0 1 0

0 1 1 0 1 1 0 1

0 0 0 1 0 1 0 1

0 0 0 1 0 0 1 1

0 1 1 0 0 1 1 0

0 1 1 0 0 0 1 1

0 0 0 1 1 0 0 0

0 0 0 1 1 1 0 0

0 0 0 1 1 1 0 1

0 1 1 0 1 1 1 0

0 1 1 0 1 1 1 0

0 1 1 0 1 1 1 0

rank_6(9)

110
```
Recap: Wavelet Trees

![Wavelet Tree Diagram]

The diagram illustrates a wavelet tree with keys and corresponding ranks. The tree is divided into intervals, and each node represents a segment of the key set. The ranks at the nodes help in querying the tree efficiently. The diagram shows intervals such as [0, 3], [0, 7], [4, 7], [0, 1], [2, 3], [4, 5], and [6, 7].

The ranks are indicated with node labels, such as rank(9) = 110, which shows how many keys are present up to the input value 9 in each interval.
Recap: Wavelet Trees

Wavelet Trees are data structures used for text indexing. The diagram shows a Wavelet Tree for the range [0, 7]. The tree is built by recursively dividing the range into smaller intervals, and each node represents a sub-range of the original range.

The tree is constructed as follows:

- The root node represents the range [0, 7].
- The root node is divided into two children nodes, one for the range [0, 3] and one for the range [4, 7].
- The range [0, 3] is divided into two children nodes, one for the range [0, 1] and one for the range [2, 3].
- The range [4, 7] is divided into two children nodes, one for the range [4, 5] and one for the range [6, 7].
- Each node stores the rank function values for its sub-range.

The rank function, \( \text{rank}_k(i) \), returns the number of elements less than or equal to \( k \) in the sub-range represented by the node.

In the example, \( \text{rank}_6(9) \) is computed for the range [0, 7]. The result is 110.
Recap: Wavelet Trees

![Wavelet Trees Diagram](image-url)
Recap: Wavelet Trees
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]

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

[0, 1]

\[
\begin{array}{cc}
0 & 1 & 1 \\
0 & 1 & 1 \\
\end{array}
\]

[2, 3]

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

[4, 5]

\[
\begin{array}{cc}
5 & 4 \\
1 & 0 \\
\end{array}
\]

[6, 7]

\[
\begin{array}{cc}
6 & 7 & 6 \\
0 & 1 & 0 \\
\end{array}
\]
Recap: Compressed Wavelet Trees

- 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

- 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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>7</th>
<th>1</th>
<th>5</th>
<th>4</th>
<th>2</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 $O(k \log \frac{n}{k}) + o(n)$ 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
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 $s' = s^2$

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
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 $s' = s^2$

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 \( O(k \log \frac{n}{k}) + o(n) \) 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 position in lookup-table
- concatenate all codes
Array \textit{SBlock}

- for every super-block \( i \), \textit{SBlock}[i] \) contains position of encoding of first block in \( i \)-th super-block in \( V \)
- \( \lceil \log n \rceil \) 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
### 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.
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
- $|V| = \sum_{i=1}^{\lceil \frac{n}{s} \rceil} \lceil \lg \binom{s}{k_i} \rceil \leq \lg \binom{n}{k} + n/s \leq \lg((n/k)^k) + n/s = k \lg \frac{n}{k} + O(\frac{n}{\lg n})$ bits
Recap: Backwards Search in the BWT

Function `BackwardsSearch(P[1..n], C, rank)

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

- 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 \textit{rank}-function
- \textit{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 \( O(occ + \lg n) \) 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 $\sigma$, 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 $O(\text{occ} + \lg n)$ time.

Space Requirements
- wavelet tree: $n \lceil \lg \sigma \rceil (1 + o(1))$ bits
- C-array: $\sigma \lceil \lg n \rceil$ 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 $s = \lg\sigma \cdot n$
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 \( BWT \)

\( \text{Definition: Run (simplified, recap)} \)

Given a text \( T \) of length \( n \), we call its substring \( T[i..j] \) a \textbf{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.)
Motivation: $r$-Index

- next: compressed index
- how to measure compressibility?
Motivation: \( r \)-Index

- next: compressed index
- how to measure compressibility?

Measure for Compressibility

- \( k \)-th order empirical entropy \( H_k \)
- number of LZ factors \( z \)
- number of BWT runs \( r \)
Motivation: $r$-Index

- next: compressed index
- how to measure compressibility?

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 \cite{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 $r \in O(z \lg n)$

more details in next lecture
Motivation: $r$-Index

- next: compressed index
- how to measure compressibility?

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
Motivation: $r$-Index

- next: compressed index
- how to measure compressibility?

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 \textit{BackwardsSearch}(P[1..n], C, rank):

1. \hspace{1em} s = 1, \hspace{1em} e = n
2. \hspace{1em} for \hspace{1em} i = m, \ldots, 1 \hspace{1em} do
3. \hspace{3em} s = C[P[i]] + rank_{P[i]}(s - 1) + 1
4. \hspace{3em} e = C[P[i]] + rank_{P[i]}(e)
5. \hspace{3em} if \hspace{1em} s > e \hspace{1em} then
6. \hspace{5em} return 0
7. \hspace{1em} return [s, e]

Goals

- simulate \textit{BWT} and \textit{rank} on \textit{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.

- **Array $I_i$** stores the position of the $i$-th run in $BWT$.
- **Array $L'_i$** stores the character of the $i$-th run in $BWT$.
- **Array $R$** builds wavelet tree for $L'_i$.
- **Array $C'$** contains the lengths of $BWT$ runs stably sorted by the runs' characters and accumulates for each character by performing exclusive prefix sum over run lengths.
- **Array $C'_\alpha$** stores the start of the run lengths in $R$ for each character $\alpha \in \Sigma$ starting at 0.
- **Bit Vector $B$** is a compressed bit vector of length $n$ containing ones at positions where $BWT$ runs start.
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 [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
- rank-support
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’

**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 $s$ before the $j$-th run ($k = R[k + \ell]$)
- compute character $\beta$ of run ($\beta = L'[j]$)
- if $\alpha \neq \beta$ return $k \oplus i$ is not in the run
- else return $k + i - l[j] + 1 \oplus i$ 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

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

\[
O(m \lg \sigma)
\]

- 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

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
- mark them in bit vector and store additional information

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
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?

- **Suffix Tree**: 1973
- **Suffix Array**: 1993
- **LCP Array**: 1993
- **BWT**: 1994
- **Wavelet Tree**: 2000
- **FM-Index**: 2000

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

