Text Indexing

Lecture 05: Burrows-Wheeler Transform

Florian Kurpicz
Recap: Text-Compression

Definition: LZ77 Factorization \([\text{ZL77}]\)

Given a text \(T\) of length \(n\) over an alphabet \(\Sigma\), the \textit{LZ77 factorization} is

- a set of \(z\) factors \(f_1, f_2, \ldots, f_z \in \Sigma^+\), such that
- \(T = f_1 f_2 \ldots f_z\) and for all \(i \in [1, z]\) \(f_i\) is
- single character not occurring in \(f_1 \ldots f_{i-1}\) or
- longest substring occurring \(\geq 2\) times in \(f_1 \ldots f_i\)

\[ T = \text{abababbbbaba}\$
- \(f_1 = a\)
- \(f_2 = b\)
- \(f_3 = abab\)
- \(f_4 = bbb\)
- \(f_5 = aba\)
- \(f_6 = \$\)

Definition: LZ78 Factorization \([\text{ZL78}]\)

Given a text \(T\) of length \(n\) over an alphabet \(\Sigma\), the \textit{LZ78 factorization} is

- a set of \(z\) factors \(f_1, f_2, \ldots, f_z \in \Sigma^+\), such that
- \(T = f_1 f_2 \ldots f_z, f_0 = \epsilon\) and for all \(i \in [1, z]\)
- if \(f_1 \ldots f_{i-1} = T[1..j - 1]\), then \(f_i\) is the longest prefix of \(T[j..n]\), such that

\[ \exists k \in [0, i), \alpha \in \Sigma \cup \{$\} : f_k = f_i \alpha\]

\[ T = \text{abababbbbaba}\$
- \(f_1 = a\)
- \(f_2 = b\)
- \(f_3 = ab\)
- \(f_4 = abb\)
- \(f_5 = bb\)
- \(f_6 = aba\)
- \(f_7 = \$\)
Definition: Burrows-Wheeler Transform

Given a text $T$ of length $n$ and its suffix array $SA$, for $i \in [1, n]$ the Burrows-Wheeler transform is

$$BWT[i] = \begin{cases} T[SA[i] - 1] & \text{if } SA[i] > 1 \\ \$ & \text{if } SA[i] = 1 \end{cases}$$

- character before the suffix in $SA$-order
- choose characters cyclic \$ for first suffix
- can compute $BWT$ in $O(n)$ time
- for binary alphabet $O(n/\sqrt{\log n})$ time and $O(n/\log n)$ words space is possible [KK19]

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
</table>
| $T$ | a | b | a | b | c | a | b | c | a | b | b | a | $\$
| $SA$ | 13 | 12 | 1 | 9 | 6 | 3 | 11 | 2 | 10 | 7 | 4 | 8 | 5 |
| $LCP$ | 0 | 0 | 1 | 2 | 2 | 5 | 0 | 2 | 1 | 1 | 4 | 0 | 3 |
| $BWT$ | a | b | $\$ | c | c | b | b | a | a | a | a | b | b |

- definition is not very descriptive
- easy way to compute $BWT$
- what can we do with the $BWT$
- PINGO can the BWT be reversed?
**Burrows-Wheeler Transform (2/2)**

**Definition: Cyclic Rotation**

Given a text $T$ of length $n$, the $i$-th cyclic rotation is

$$T(i) = T[i..n]T[1..i]$$

- $i$-th cyclic rotation is concatenation of $i$-th suffix and $(i - 1)$-th prefix

**Definition: Burrows-Wheeler Transform (alt.)**

Given a text $T$ and a matrix containing all its cyclic rotations in lexicographical order as columns, then the Burrows-Wheeler transform of the text is the last row of the matrix

$$T = ababcabcabba$$

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>$</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>$</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>a</td>
<td>$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>

$T = ababcabcabba$
two important rows in the matrix
other rows are not needed at all
there is a special relation between the two rows

Later this lecture

First Row\( F \)
- contains all characters or the text in sorted order

Last Row\( L \)
- is the \textit{BWT} itself
Properties of the BWT: Rank of Characters

**Definition: Rank**

Given a text $T$ over an alphabet $\Sigma$, the **rank** of a character at position $i \in [1, n]$ is

$$\text{rank}(i) = |\{j \in [1, i]: T[i] = T[j]\}|$$

- **rank** is the number of same characters that occur before in the text.
- **mark** ranks of characters w.r.t. text not BWT.
- **order** of ranks is the same in first and last row.

**Example:**

- $T = \text{ababcabcabba}\$
- $T(1) = \text{a}$, $T(2) = \text{b}$, $T(3) = \text{a}$, $T(4) = \text{b}$, $T(5) = \text{a}$, $T(6) = \text{b}$, $T(7) = \text{a}$, $T(8) = \text{c}$, $T(9) = \text{b}$, $T(10) = \text{a}$, $T(11) = \text{b}$, $T(12) = \text{a}$, $T(13) = \text{b}$, $T(14) = \text{b}$, $T(15) = \text{a}$, $T(16) = \text{b}$.

**Ranking Example:**

- $\text{rank}(1) = \{\}$
- $\text{rank}(2) = \{1\}$
- $\text{rank}(3) = \{1, 2\}$
- $\text{rank}(4) = \{1, 2, 3\}$
- $\text{rank}(5) = \{1, 2, 3, 4\}$
- $\text{rank}(6) = \{1, 2, 3, 4, 5\}$
- $\text{rank}(7) = \{1, 2, 3, 4, 5, 6\}$
- $\text{rank}(8) = \{1, 2, 3, 4, 5, 6, 7\}$
- $\text{rank}(9) = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $\text{rank}(10) = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $\text{rank}(11) = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- $\text{rank}(12) = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$
- $\text{rank}(13) = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- $\text{rank}(14) = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}$
- $\text{rank}(15) = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$
- $\text{rank}(16) = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$

**Diagram:**

- The diagram illustrates the ranking of characters in the text $T = \text{ababcabcabba}$
- Each node represents a character and its rank is indicated by the number associated with it.
- The edges connect characters based on their rank, showing the order in which characters occur in the text.

**Table:**

<table>
<thead>
<tr>
<th>Character</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>b</td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
</tr>
</tbody>
</table>
**LF-Mapping (1/2)**

- want to map characters from last to first row
- why do we want this?
  - helps with pattern matching
  - transform BWT back to $T$

**Definition: LF-mapping**

Given a text $T$ of length $n$ and its suffix array $SA$, then the LF-mapping is a permutation of $[1, n]$, such that

$$LF(i) = j \iff SA[j] = SA[i] - 1$$

- similar to definition of BWT
- requires $SA$ or explicitly saving LF-mapping
**Definition: C-Array and Rank-Function**

Given a text $T$ of length $n$ over an alphabet $\Sigma$, $\alpha \in \Sigma$, and $i \in [1, n]$ then

$$C[\alpha] = |i \in [1, n]: T[i] < \alpha|$$

and

$$\text{rank}_\alpha(i) = |\{j \in [1, i]: T[j] = \alpha\}|$$

- $C$ contains total number of smaller characters
- $\text{rank}_\alpha$ contains number of $\alpha$'s in prefix $T[1..i]$
- $\text{rank}_\alpha$ can be computed in $O(1)$ time [FM00]

**LF-Mapping (2/2)**

Given a $BWT$, its $C$-array, and its $\text{rank}$-Function, then

$$LF(i) = C[BWT[i]] + \text{rank}_{BWT[i]}(i)$$

- rank now on $BWT$
- $C$ is exclusive prefix sum over histogram
Reversing the BWT (1/2)

- characters (w.r.t. text) preserve order in L and F
- LF-mapping returns previous character in text
Reversing the BWT (2/2)

- Characters (w.r.t. text) preserve order in \( L \) and \( F \)
- \( LF \)-mapping returns previous character in text

- \( T[n] = \$ \) and \( T^{(n)} \) in first row
- Apply \( LF \)-mapping on result to obtain any character

\[
T[n - i] = L[LF(LF(\ldots (LF(1))\ldots))]^{i-1 \text{ times}}
\]

<table>
<thead>
<tr>
<th>( L )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>0</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( b )</td>
<td>$</td>
<td>( c )</td>
<td>( c )</td>
<td>( b )</td>
<td>( b )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( b )</td>
<td>( b )</td>
<td>( b )</td>
<td></td>
</tr>
<tr>
<td>( LF )</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>12</td>
<td>13</td>
<td>8</td>
<td>9</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

- \( T[13] = \$ \) and \( k = 1 \)
- \( T[12] = L[1] = a \) and \( k = LF(1) = 2 \)
- \( T[11] = L[2] = b \) and \( k = LF(2) = 7 \)
- \( T[10] = L[7] = b \) and \( k = LF(7) = 9 \)
- \( T[9] = L[9] = a \) and \( k = LF(9) = 4 \)
- \( T[9] = L[4] = c \) and \( k = LF(4) = 12 \)
- \( \ldots \)
Properties of the BWT: Runs

- **BWT** is reversible
- can be used for lossless compression

**Definition: Run (simplified)**

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 \( T[j + 1] \neq T[j] \)

พา (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.)

- **BWT** contains lots of runs
- same context is often grouped together

\[
\begin{array}{ccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 0 & 11 & 12 & 13 \\
\text{L} & a & b & $ & c & c & b & b & a & a & a & a & b & b
\end{array}
\]
Compressing the BWT: Run-Length Compression

Definition: Run-Length Encoding

Given a text $T$, represent each run $T[i..i + \ell)$ as tuple $(T[i], \ell)$

$T = \text{ababcabcabba}$

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>0</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>BWT</td>
<td>a</td>
<td>b</td>
<td>$</td>
<td>$</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

- $(a, 1)$
- $(b, 1)$
- $(\$, 1)$
- $(c, 2)$
- $(b, 2)$
- $(a, 4)$
- $(b, 2)$
Compressing the BWT: Move-to-Front

Definition: Move-To-Front Encoding

Given a text \( T \) over an alphabet \( \Sigma = [1, \sigma] \), the MTF encoding \( MTF(T) \) of the text is computed as follows:

- Start with a list \( X = \Sigma[1], \Sigma[2], \ldots, \Sigma[\sigma] \).
- Scan text from left to right, for character \( T[i] \):
  - Append position of \( T[i] \) in \( X \) to \( MTF(T) \) and
  - Move \( T[i] \) to front of \( X \).

MTF encoding can easily be reverted.

- Consists of many small numbers.
- Runs are preserved.
- Use Huffman on encoding.

MTF encoding has no theoretical improvement but is good in practice.

\[ T = \text{ababcabcabba$} \]

<table>
<thead>
<tr>
<th>BWT</th>
<th>1 2 3 4 5 6 7 8 9 0 11 12 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b $ c c b b a a a a b b</td>
</tr>
</tbody>
</table>

- \( X = $, a, b, c \)
- \( MTF = 2 \) and \( X = a, $, b, c \)
- \( MTF = 2 3 \) and \( X = b, a, $, c \)
- \( MTF = 2 3 3 \) and \( X = $, b, a, c \)
- \( MTF = 2 3 3 4 \) and \( X = c, $, b, a \)
- \( MTF = 2 3 3 4 1 \) and \( X = c, $, b, a \)
- \( MTF = 2 3 3 4 1 1 \) and \( X = c, $, b, a \)
- \( \ldots \)
- \( MTF = 2 3 3 4 1 1 3 1 4 1 1 1 2 1 \)
Pattern Matching using the BWT

Recap

Given a text $T$ of length $n$ over an alphabet $\Sigma$, $\alpha \in \Sigma$, and $i \in [1, n]$ then

$$C[\alpha] = |i \in [1, n]: T[i] < \alpha|$$

and

$$\text{rank}_\alpha(i) = |\{j \in [1, i]: T[j] = \alpha\}|$$

- interval for $\alpha$ is $[C[\alpha - 1], C[\alpha + 1]]$
- find sub-interval using $\text{rank}_\alpha$
- example on the board

- find interval of occurrences in $SA$ using $BWT$
- $SA$ is divided into intervals based on first character of suffix as seen during SAIS
- text from $BWT$ is backwards
- search pattern backwards
Backwards Search in the BWT

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 $[s, e]$

- 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
Sampling the Suffix Array

- Reporting queries require SA
- Storing whole SA requires too much space
- Better: sample every \( s \)-th SA position in \( SA' \)

- How to find sampled position?
- Mark sampled positions in bit vector of size \( n \)
- If match occurs, check if position is sampled
- Otherwise, find sample using LF
- If \( SA[i] = j \) then \( SA[LF(i)] = j - 1 \)

- \( rank_1(i) \) in bit vector is number of sample
- \( SA'[rank_1(i)] \) is sampled value
- \( SA'[rank_1(i)] + \# \) steps till sample found is correct SA value

- Finding a sample requires \( O(s \cdot t_{rank}) \) time
Efficient Bit Vectors in Practice (1/3)

- **std::vector<char/int/...>**
  - easy access
  - very big: 1, 4, ... bytes per bit

- **std::vector<bool>**
  - bit vector in C++ (1 bit per byte)
  - easy access
  - layout depending on implementation

- **std::vector<uint64_t>**
  - requires 8 bytes per bit (?)
  - store 64 bits in 8 bytes
  - how to access bits

- **i/64** is position in 64-bit word
- **i%64** is position in word

![Diagram showing layout of bit vector]
// There is a bit vector
std::vector<uint64_t> bit_vector;

// access i-th bit
uint64_t block = bit_vector[i/64];
bool bit = (block >> (63 - (i % 64))) & 1ULL;
### Efficient Bit Vectors in Practice (3/3)

**Fill bit vector from left to right**

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>...</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

| 0   | 0   | 0   | 0   | 0   | ... | 1 | 0 |

**Assembler code:**

- `mov ecx, edi`
- `not ecx`
- `shr rsi, cl`
- `mov eax, esi`
- `and eax, 1`

---

**Fill bit vector right to left**

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>...</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>...</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

| 0   | 0   | ... | 1 | 1 | 0 | 0 | 1 | 0 |

**Assembler code:**

- `mov ecx, edi`
- `shr rsi, cl`
- `mov eax, esi`
- `and eax, 1`
Rank Queries in Bit Vectors (1/2)

\[
\begin{align*}
\text{rank}_\alpha(i) & \quad \text{\# of } \alpha \text{ s before } i \\
\text{select}_\alpha(j) & \quad \text{position of } j\text{-th } \alpha
\end{align*}
\]

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

- \text{rank}_0(5) = 2
- \text{select}_1(5)

- \# of 0s w.r.t. BV
- \# of 0s w.r.t. super-block
Rank Queries in Bit Vectors (2/2)

- for a bit vector of size $n$
  - blocks of size $s = \left\lfloor \frac{\lg n}{2} \right\rfloor$
  - super blocks of size $s' = s^2 = \Theta(\lg^2 n)$

- for all $\left\lfloor \frac{n}{s'} \right\rfloor$ super blocks, store number of 0s from beginning of super block to end of block
  - $\frac{n}{s'} \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ bits of space

PINGO how fast can rank queries be answered?

- for all $\left\lfloor \frac{n}{s} \right\rfloor$ blocks, store number of 0s from beginning of super block to end of block
  - $\frac{n}{s} \cdot \lg s = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ bits of space

- for all length-$s$ bit vectors, for every position $i$
  - store number of 0s up to $i$
  - $2^{\frac{\lg n}{2}} \cdot s \cdot \lg s = O(\sqrt{n} \lg n \lg \lg n) = o(n)$ bits of space

- query in $O(1)$ time
  - $rank_0(i) = i - rank_1(i)$
The FM-Index (First Look) [FM00]

Building Blocks of FM-Index
- wavelet tree on BWT providing rank-function
  - wavelet trees are topic of next lecture!
- C-array
- sampled suffix array with sample rate $s$
- bit vector marking sampled suffix array positions

Lemma: FM-Index Space Requirements
Given a text $T$ of length $n$ over an alphabet of size $\sigma$, the FM-index requires $O(n \lg \sigma)$ bits of space

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 $s = \lceil \lg \sigma \rceil n$
Conclusion and Outlook

This Lecture
- Burrows-Wheeler transform
- introduction to FM-index
- efficient bit vectors
- rank queries on bit vectors

Next Lecture
- wavelet trees
- more on FM-index

Linear Time Construction

ST -> SA
LZ -> LCP
BWT
Bibliography I


