Text Indexing

Lecture 05: Text-Compression

Florian Kurpicz
Recap: Suffix Array and LCP-Array

**Definition: Suffix Array** [GBS92; MM93]

Given a text $T$ of length $n$, the suffix array (SA) is a permutation of $[1..n]$, such that for $i \leq j \in [1..n]$

$$T[SA[i]..n] \leq T[SA[j]..n]$$

**Definition: Longest Common Prefix Array**

Given a text $T$ of length $n$ and its SA, the LCP-array is defined as

$$LCP[i] = \begin{cases} 0 & i = 1 \\ \max\{\ell : T[SA[i]..SA[i] + \ell) = T[SA[i-1]..SA[i-1] + \ell)\} & i \neq 1 \end{cases}$$
Why Compression

Types of Compression

- lossy compression
  - audio, video, pictures, ...
- lossless compression
  - audio, text, ...

This Lecture

measure compressibility
different compression algorithms
both types
space/time requirements of compression
make use of known concepts
Why Compression

Types of Compression

- lossy compression
  - audio, video, pictures, ...

- lossless compression
  - audio, text, ...

- only interested in lossless compression
- faster data transfer
- cheaper storage costs
- “compress once, decompress often”
Why Compression

Types of Compression
- lossy compression
  - audio, video, pictures, ...
- lossless compression
  - audio, text, ...

only interested in lossless compression
- faster data transfer
- cheaper storage costs
- “compress once, decompress often”

Types of Text-Compression
- entropy coding (compress characters)
- dictionary compression (compress substrings)
- ...
Why Compression

Types of Compression
- lossy compression
  - audio, video, pictures, ...
- lossless compression
  - audio, text, ...
- only interested in lossless compression
- faster data transfer
- cheaper storage costs
- “compress once, decompress often”

Types of Text-Compression
- entropy coding
  - compress characters
- dictionary compression
  - compress substrings
  - ...

This Lecture
- measure compressibility
- different compression algorithms
  - both types
- space/time requirements of compression algorithms
- make use of known concepts
**k-th Order Empirical Entropy [KM99] (1/2)**

**Definition: Histogram**

Given a text $T$ of length $n$ over an alphabet of size $\sigma$, a histogram $Hist[1..\sigma]$ is defined as

$$Hist[i] = |\{j \in [1, n]: T[j] = i\}|$$
Definition: Histogram
Given a text $T$ of length $n$ over an alphabet of size $\sigma$, a histogram $Hist[1..\sigma]$ is defined as

$$Hist[i] = |\{ j \in [1, n] : T[j] = i \}|$$

Definition: 0-th Order Empirical Entropy
Given a text $T$ of length $n$ over an alphabet $\Sigma = [1, \sigma]$ and its histogram $Hist$, then

$$H_0(T) = (1/n) \sum_{i=1}^{\sigma} Hist[i] \lg(n/\text{Hist}[i])$$
**k-th Order Empirical Entropy [KM99](1/2)**

**Definition: Histogram**
Given a text \( T \) of length \( n \) over an alphabet of size \( \sigma \), a histogram \( \text{Hist}[1..\sigma] \) is defined as

\[
\text{Hist}[i] = |\{ j \in [1, n]: T[j] = i \}|
\]

**Definition: 0-th Order Empirical Entropy**
Given a text \( T \) of length \( n \) over an alphabet \( \Sigma = [1, \sigma] \) and its histogram \( \text{Hist} \), then

\[
H_0(T) = \frac{1}{n} \sum_{i=1}^{\sigma} \text{Hist}[i] \log(n/\text{Hist}[i])
\]

- \( T = \text{abbaacaaaba}$
- \( n = 12 \)
- \( \text{Hist}[a] = 7 \)
- \( \text{Hist}[b] = 3 \)
- \( \text{Hist}[c] = 1 \)
- \( \text{Hist}[$] = 1 \)

\[
H_0(T) = \frac{1}{12} (7 \log(12/7) + 3 \log(12/3) + 1 \log(12)) \approx 1.55
\]
Definition: Histogram

Given a text $T$ of length $n$ over an alphabet of size $\sigma$, a histogram $Hist[1..\sigma]$ is defined as

$$Hist[i] = |\{j \in [1, n]: T[j] = i\}|$$

Definition: 0-th Order Empirical Entropy

Given a text $T$ of length $n$ over an alphabet $\Sigma = [1, \sigma]$ and its histogram $Hist$, then

$$H_0(T) = \frac{1}{n} \sum_{i=1}^{\sigma} Hist[i] \log(n/Hist[i])$$

Example:

- $T = \text{abbaaacaaba}$
- $n = 12$
- $Hist[a] = 7$
- $Hist[b] = 3$
- $Hist[c] = 1$
- $Hist[$ = 1
**Definition: Histogram**

Given a text $T$ of length $n$ over an alphabet of size $\sigma$, a histogram $\text{Hist}[1..\sigma]$ is defined as

\[ \text{Hist}[i] = |\{j \in [1, n]: T[j] = i\}| \]

**Definition: 0-th Order Empirical Entropy**

Given a text $T$ of length $n$ over an alphabet $\Sigma = [1, \sigma]$ and its histogram $\text{Hist}$, then

\[ H_0(T) = \frac{1}{n} \sum_{i=1}^{\sigma} \text{Hist}[i] \log(n/\text{Hist}[i]) \]

Example:

- $T = \text{abbaacaaba}$
- $n = 12$
- $\text{Hist}[a] = 7$
- $\text{Hist}[b] = 3$
- $\text{Hist}[c] = 1$
- $\text{Hist}[$ = 1

\[ H_0(T) = \frac{1}{12} (7 \log(12/7) + 3 \log(12/3) + 1 \log(12/1) + 1 \log(12/1)) \approx 1.55 \]
Given a text $T$ over an alphabet $\Sigma$ and a string $S \in \Sigma^k$, $T_S$ the concatenation of all characters that occur in $T$ after $S$ in text order.

- $T = \text{abcdabcdeabcd}$
- $S = \text{abc}$
- $T_S = \text{ded}$

**Definition: $k$-th Order Empirical Entropy**

Given a text $T$ of length $n$ over an alphabet $\Sigma = [1, \sigma]$ and its histogram $\text{Hist}$, then

$$H_k = \frac{1}{n} \sum_{S \in \Sigma^k} |T_S| \cdot H_0(T_S)$$
**k-th Order Empirical Entropy (2/2)**

Given a text $T$ over an alphabet $\Sigma$ and a string $S \in \Sigma^k$, $T_S$ the concatenation of all characters that occur in $T$ after $S$ in text order

- $T = \text{abcdabceabcd}$
- $S = \text{abc}$
- $T_S = \text{ded}$

**Definition: k-th Order Empirical Entropy**

Given a text $T$ of length $n$ over an alphabet $\Sigma = [1, \sigma]$ and its histogram $\text{Hist}$, then

$$H_k = \frac{1}{n} \sum_{S \in \Sigma^k} |T_S| \cdot H_0(T_S)$$

PINGO can we describe a property of $H_k$
### Example for $k$-th Order Empirical Entropy [Kur20]

<table>
<thead>
<tr>
<th>Name</th>
<th>$\sigma$</th>
<th>$n$</th>
<th>$H_0$</th>
<th>$H_1$</th>
<th>$H_2$</th>
<th>$H_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commoncrawl</td>
<td>243</td>
<td>196,885,192,752</td>
<td>6.19</td>
<td>4.49</td>
<td>2.52</td>
<td>2.08</td>
</tr>
<tr>
<td>DNA</td>
<td>4</td>
<td>218,281,833,486</td>
<td>1.99</td>
<td>1.97</td>
<td>1.96</td>
<td>1.95</td>
</tr>
<tr>
<td>Proteins</td>
<td>26</td>
<td>50,143,206,617</td>
<td>4.21</td>
<td>4.20</td>
<td>4.19</td>
<td>4.17</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>213</td>
<td>246,327,201,088</td>
<td>5.38</td>
<td>4.15</td>
<td>3.05</td>
<td>2.33</td>
</tr>
<tr>
<td>SuffixArrayCC</td>
<td>$n$</td>
<td>137,438,953,472</td>
<td>37 ($= \lg n$)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RussianWordBased</td>
<td>29</td>
<td>9,232,978,762</td>
<td>10.93</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
Example for $\kappa$-th Order Empirical Entropy [Kur20]

<table>
<thead>
<tr>
<th>Name</th>
<th>$\sigma$</th>
<th>$n$</th>
<th>$H_0$</th>
<th>$H_1$</th>
<th>$H_2$</th>
<th>$H_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commoncrawl</td>
<td>243</td>
<td>196,885,192,752</td>
<td>6.19</td>
<td>4.49</td>
<td>2.52</td>
<td>2.08</td>
</tr>
<tr>
<td>DNA</td>
<td>4</td>
<td>218,281,833,486</td>
<td>1.99</td>
<td>1.97</td>
<td>1.96</td>
<td>1.95</td>
</tr>
<tr>
<td>Proteins</td>
<td>26</td>
<td>50,143,206,617</td>
<td>4.21</td>
<td>4.20</td>
<td>4.19</td>
<td>4.17</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>213</td>
<td>246,327,201,088</td>
<td>5.38</td>
<td>4.15</td>
<td>3.05</td>
<td>2.33</td>
</tr>
<tr>
<td>SuffixArrayCC</td>
<td>$n$</td>
<td>137,438,953,472</td>
<td>37 ($= \lg n$)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RussianWordBased</td>
<td>29,263</td>
<td>9,232,978,762</td>
<td>10.93</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

- does not measure repetitions well
- there are other measures
Huffman Coding [Huf52]

- idea is to create a binary tree
- each character $\alpha$ is a leaf and has weight $\text{Hist}[\alpha]$
- create node for two nodes without parent with smallest weight
- give new node total weight of children
- repeat until only one node without parent remains

- label edges:
  - left edge: 0
  - right edge: 1
- path to children gives code for character

$$T = \text{cbcacaa}$$
Huffman Coding \([\text{Huf52}]\)

- Idea is to create a binary tree
- Each character \(\alpha\) is a leaf and has weight \(\text{Hist}[\alpha]\)
- Create node for two nodes without parent with smallest weight
- Give new node total weight of children
- Repeat until only one node without parent remains

- Label edges:
  - Left edge: 0
  - Right edge: 1
- Path to children gives code for character

- Codes are variable length and prefix-free
- Tree/dictionary needed for decoding

\[ T = \text{cbcacaa} \]

\[ \begin{aligned} \{a, b, c\} & : 7 \\ \{a, b\} & : 4 \\ \{a\} & : 3 \\ \{b\} & : 1 \\ \{c\} & : 3 \end{aligned} \]
Canonical Huffman Coding

- start with Huffman codes, code word 0, and length 1
- to get canonical code for current length, then add 1 to code word
- to update length add 1 and append required amount of zeros to code word
Canonical Huffman Coding

- start with Huffman codes, code word 0, and length 1
- to get canonical code for current length, then add 1 to code word
- to update length add 1 and append required amount of zeros to code word

Continue From Last Slide

- length 1: c
- length 2: a, b
Canonical Huffman Coding

- start with Huffman codes, code word 0, and length 1
- to get canonical code for current length, then add 1 to code word
- to update length add 1 and append required amount of zeros to code word

Continue From Last Slide

- length 1: c
- length 2: a, b
- start with 0 → code for c
Canonical Huffman Coding

- start with Huffman codes, code word 0, and length 1
- to get canonical code for current length, then add 1 to code word
- to update length add 1 and append required amount of zeros to code word

Continue From Last Slide

- length 1: c
- length 2: a, b
- start with 0 → code for c
- add 1 and append 0
Canonical Huffman Coding

- Start with Huffman codes, code word 0, and length 1
- To get canonical code for current length, then add 1 to code word
- To update length add 1 and append required amount of zeros to code word

Continue From Last Slide

- Length 1: c
- Length 2: a, b
- Start with 0 → code for c
- Add 1 and append 0
- 10 → code for a
start with Huffman codes, code word 0, and length 1

to get canonical code for current length, then add 1 to code word

to update length add 1 and append required amount of zeros to code word

Continue From Last Slide

- length 1: c
- length 2: a, b
- start with 0 → code for c
- add 1 and append 0
- 10 → code for a
- add 1
Canonical Huffman Coding

- start with Huffman codes, code word 0, and length 1
- to get canonical code for current length, then add 1 to code word
- to update length add 1 and append required amount of zeros to code word

Continue From Last Slide

- length 1: c
- length 2: a, b
- start with 0 → code for c
- add 1 and append 0
- 10 → code for a
- add 1
- 11 → code for b
Canonical Huffman Coding

- start with Huffman codes, code word 0, and length 1
- to get canonical code for current length, then add 1 to code word
- to update length add 1 and append required amount of zeros to code word

- all codes of same length are increasing
- required for Huffman-shaped wavelet trees

Continue From Last Slide

- length 1: c
- length 2: a, b
- start with 0 → code for c
- add 1 and append 0
- 10 → code for a
- add 1
- 11 → code for b

* will be discussed in a later lecture
Canonical Huffman Coding

- Start with Huffman codes, code word 0, and length 1.
- To get canonical code for current length, then add 1 to code word.
- To update length add 1 and append required amount of zeros to code word.

- All codes of same length are increasing.
- Required for Huffman-shaped wavelet trees.

Continue From Last Slide

- Length 1: c
- Length 2: a, b
- Start with 0 → code for c
- Add 1 and append 0
- 10 → code for a
- Add 1
- 11 → code for b

- Still variable length and prefix-free.
- Instead of tree only require lengths’ of codes and corresponding characters.

*Note: will be discussed in a later lecture*
- start with Huffman codes, code word 0, and length 1
- to get canonical code for current length, then add 1 to code word
- to update length add 1 and append required amount of zeros to code word

- all codes of same length are increasing
- required for Huffman-shaped wavelet trees
- will be discussed in a later lecture

**PINGO** what are some advantages of canonical Huffman codes?

- still variable length and prefix-free
- instead of tree only require lengths’ of codes and corresponding characters
Shannon-Fano Coding \([\text{Fan49}; \text{Sha48}]\)

- given a text \(T\) of length \(n\) over an alphabet \(\Sigma\) and its histogram \(\text{hist}\)
- each character \(\alpha \in \Sigma\) receives a code of length
  \[ \ell_\alpha = \lceil \lg \frac{n}{\text{Hist}[\alpha]} \rceil \]
Shannon-Fano Coding \cite{Fan49; Sha48}

- given a text $T$ of length $n$ over an alphabet $\Sigma$ and its histogram $\text{hist}$
- each character $\alpha \in \Sigma$ receives a code of length
  \[ \ell_{\alpha} = \lceil \lg \frac{n}{\text{Hist}[\alpha]} \rceil \]

- show that there always exists such a code
- assume a complete binary tree of depth
  \[ \ell_{\text{max}} = \max_{\alpha \in \Sigma} \ell_{\alpha} \] with all free nodes
- left edges labeled 0, right edges labeled 1
- characters ordered by frequency
  \[ (\ell_1 \geq \ell_2 \geq \cdots \geq \ell_{\sigma}) \]
- assign characters the leftmost free node
- mark all nodes above and below as non-free
Shannon-Fano Coding [Fan49; Sha48]

- A text $T$ of length $n$ over an alphabet $\Sigma$ and its histogram $\text{hist}$
- Each character $\alpha \in \Sigma$ receives a code of length
  $$\ell_\alpha = \lceil \lg \frac{n}{\text{Hist}[\alpha]} \rceil$$

Proof there are enough free nodes (Sketch)

- A code $\ell_\alpha$ marks $2^{\ell_{\text{max}} - \ell_\alpha}$ nodes
- Total number of marked leafs is
  $$\sum_{\alpha \in \Sigma} 2^{\ell_{\text{max}} - \ell_\alpha} = 2^{\ell_{\text{max}}} \sum_{\alpha \in \Sigma} 2^{-\ell_\alpha}$$
  $$\leq 2^{\ell_{\text{max}}} \sum_{\alpha \in \Sigma} 2^{-\lceil \lg \frac{n}{\text{Hist}[\alpha]} \rceil}$$
  $$\leq 2^{\ell_{\text{max}}} \sum_{\alpha \in \Sigma} \frac{\text{Hist}[\alpha]}{n}$$
  $$\leq 2^{\ell_{\text{max}}}$$

- Show that there always exists such a code
- Assume a complete binary tree of depth $\ell_{\text{max}} = \max_{\alpha \in \Sigma} \ell_\alpha$ with all free nodes
- Left edges labeled 0, right edges labeled 1
- Characters ordered by frequency
  ($l_1 \geq l_2 \geq \cdots \geq l_\sigma$)
- Assign characters the leftmost free node
- Mark all nodes above and below as non-free
Optimality of Both

- $H_0$ gives average number of bits needed to encode character
- $nH_0(T)$ is lower bound for compression without context
Optimality of Both

- $H_0$ gives average number of bits needed to encode character
- $nH_0(T)$ is lower bound for compression without context

- one can show that no fixed-letter code can be better than Huffman
- Shannon-Fano codes can be slightly longer than Huffman
- even Shannon-Fano achieves $H_0$-compression
Optimality of Both

- $H_0$ gives average number of bits needed to encode character
- $nH_0(T)$ is lower bound for compression without context

- one can show that no fixed-letter code can be better than Huffman
- Shannon-Fano codes can be slightly longer than Huffman
- even Shannon-Fano achieves $H_0$-compression

Proof (Sketch)

- let $T$ be a text of length $n$ over an alphabet $\Sigma$ with histogram $Hist$
- let $T_{SF}$ be the Shannon-Fano encoded text
- average length of encoded character is

$$\frac{1}{n} |T_{SF}| = \left(\frac{1}{n}\right) \sum_{\alpha \in \Sigma} Hist[\alpha] \left\lceil \log \frac{n}{Hist[\alpha]} \right\rceil$$

$$\leq \sum_{\alpha \in \Sigma} \frac{Hist[\alpha]}{n} \left(\log \frac{n}{Hist[\alpha]} + 1\right)$$

$$= \sum_{\alpha \in \Sigma} \frac{Hist[\alpha]}{n} \log \frac{n}{Hist[\alpha]} + \sum_{\alpha \in \Sigma} \frac{Hist[\alpha]}{n}$$

$$= H_0(T) + 1$$
Problem with the Previous Approaches

- does not work well with repetitions
- better encode $605 \times a$
Definition: LZ77 Factorization

Given a text $T$ of length $n$ over an alphabet $\Sigma$, the **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 $f_1 \ldots f_i$
Definition: LZ77 Factorization

Given a text $T$ of length $n$ over an alphabet $\Sigma$, the 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 $f_1 \ldots f_i$

$T = \text{abababbbaba}$
Definition: LZ77 Factorization

Given a text $T$ of length $n$ over an alphabet $\Sigma$, the 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 $f_1 \ldots f_i$

Example:

$T = \text{abababbbabab}$

- $f_1 = \text{a}$

$T = \text{aaa} \ldots \text{aa}$

- $f_1 = \text{a}$

$T = \text{abababbbabab}$

- $f_3 = \text{b}$

$T = \text{aaa} \ldots \text{aa}$

- $f_3 = \text{a}$
Lempel-Ziv 77 [ZL77]

Definition: LZ77 Factorization

Given a text $T$ of length $n$ over an alphabet $\Sigma$, the 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 $f_1 \ldots f_i$

$T = abababbbaba$

- $f_1 = a$
- $f_2 = b$
Lempel-Ziv 77 \([\text{ZL77}]\)

**Definition: LZ77 Factorization**

Given a text \(T\) of length \(n\) over an alphabet \(\Sigma\), the **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 \(f_1 \ldots f_i\)

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

- \(f_1 = a\)
- \(f_2 = b\)
- \(f_3 = \text{abab}\)
Lempel-Ziv 77 [ZL77]

Definition: LZ77 Factorization

Given a text $T$ of length $n$ over an alphabet $\Sigma$, the **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 $f_1 \ldots f_i$

For example:

$T = \text{abababbbaba}$

- $f_1 = a$
- $f_2 = b$
- $f_3 = \text{abab}$
- $f_4 = \text{bbb}$
Definition: LZ77 Factorization

Given a text \( T \) of length \( n \) over an alphabet \( \Sigma \), the \textbf{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 \( f_1 \ldots f_i \)

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

- \( f_1 = a \)
- \( f_2 = b \)
- \( f_3 = \text{abab} \)
- \( f_4 = \text{bbb} \)
- \( f_5 = \text{aba} \)
Definition: LZ77 Factorization

Given a text $T$ of length $n$ over an alphabet $\Sigma$, the **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 $f_1 \ldots f_i$
Definition: LZ77 Factorization

Given a text $T$ of length $n$ over an alphabet $\Sigma$, the 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 $f_1 \ldots f_i$

$Lempel-Ziv 77 [ZL77]$
Representation of Factors

- Factors can be represented as tuple
  \[(\ell_i, p_i)\]
- \(\ell_i = 0\)
  - Factor is a single character
  - Encode character in \(p_i\)
- \(\ell_i > 0\)
  - Factor is a length-\(\ell_i\) substring
  - \(f_i = T[p_i \ldots p_i + \ell_i]\)
Representation of Factors

- Factors can be represented as tuple $(\ell_i, p_i)$

  - $\ell_i = 0$
    - Factor is a single character
    - Encode character in $p_i$
  
  - $\ell_i > 0$
    - Factor is a length-$\ell_i$ substring
    - $f_i = T[p_i..p_i + \ell_i]$

$T = \text{abababbbaba}$

- $f_1 = a$
- $f_2 = b$
- $f_3 = \text{abab}$
- $f_4 = \text{bbb}$
- $f_5 = \text{aba}$
- $f_6 = \$$
Representation of Factors

- Factors can be represented as tuple \((\ell_i, p_i)\)
- \(\ell_i = 0\)
  - Factor is a single character
  - Encode character in \(p_i\)
- \(\ell_i > 0\)
  - Factor is a length-\(\ell_i\) substring
  - \(f_i = T[p_i..p_i + \ell_i]\)

\[
T = \text{abababbbaba}\$

- \(f_1 = a = (0, a)\)
- \(f_2 = b = (0, b)\)
- \(f_3 = abab = (4, 1)\)
- \(f_4 = bbb = (3, 6)\)
- \(f_5 = aba = (3, 1) = (3, 3)\)
- \(f_6 = $ = (0, $)\)
Representation of Factors

- Factors can be represented as tuple
  \((\ell_i, p_i)\)

- \(\ell_i = 0\)
  - Factor is a single character
  - Encode character in \(p_i\)

- \(\ell_i > 0\)
  - Factor is a length-\(\ell_i\) substring
  - \(f_i = T[p_{i..i+\ell_i}]\)

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

- \(f_1 = a = (0, a)\)
- \(f_2 = b = (0, b)\)
- \(f_3 = abab = (4, 1)\)
- \(f_4 = bbb = (3, 6)\)
- \(f_5 = aba = (3, 1) = (3, 3)\)
- \(f_6 = $ = (0, $)\)
Representation of Factors

- Factors can be represented as tuple 
  \[(\ell_i, p_i)\]
- \(\ell_i = 0\)
  - Factor is a single character
  - Encode character in \(p_i\)
- \(\ell_i > 0\)
  - Factor is a length-\(\ell_i\) substring
  - \(f_i = T[p_i..p_i + \ell_i]\)

\[T = \text{abababbbaba}\$
- \(f_1 = a = (0, a)\)
- \(f_2 = b = (0, b)\)
- \(f_3 = \text{abab} = (4, 1)\)
- \(f_4 = \text{bbb} = (3, 6)\)
- \(f_5 = \text{aba} = (3, 1) = (3, 3)\)
- \(f_6 = \$ = (0, \$)\)

- Finding the right-most reference is hard
Definition: Previous and Next Smaller Value Arrays

Let $A[1..n]$ be an integer array, then

- $PSV[i] = \max\{j \in [1, i) : A[j] < A[i]\}$
- $NSV[i] = \min\{j \in (i, n] : A[j] < A[i]\}$

<table>
<thead>
<tr>
<th></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>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</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>
<td>$$$</td>
</tr>
<tr>
<td>$SA$</td>
<td>13</td>
<td>12</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>11</td>
<td>2</td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>$PSV$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>$NSV$</td>
<td>2</td>
<td>3</td>
<td>$\infty$</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>$\infty$</td>
<td>10</td>
<td>11</td>
<td>$\infty$</td>
<td>13</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$LCP$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Definition: Previous and Next Smaller Value Arrays

Let \( A[1..n] \) be an integer array, then

- \( PSV[i] = \max\{j \in [1, i) : A[j] < A[i]\} \)
- \( NSV[i] = \min\{j \in (i, n] : A[j] < A[i]\} \)

In the Context of \( SA \)

- close to the suffix in \( SA \)
- longest possible common prefix
- before the suffix in text order

<table>
<thead>
<tr>
<th>( T )</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>b</th>
<th>a</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( SA )</td>
<td>13</td>
<td>12</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>11</td>
<td>2</td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>( PSV )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>( NSV )</td>
<td>2</td>
<td>3</td>
<td>∞</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>∞</td>
<td>10</td>
<td>11</td>
<td>∞</td>
<td>13</td>
<td>∞</td>
</tr>
<tr>
<td>( LCP )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Previous and Next Smaller Values (1/2)

Definition: Previous and Next Smaller Value Arrays

Let $A[1..n]$ be an integer array, then
- $PSV[i] = \max\{j \in [1, i) : A[j] < A[i]\}$
- $NSV[i] = \min\{j \in (i, n] : A[j] < A[i]\}$

In the Context of SA
- close to the suffix in $SA$
- longest possible common prefix
- before the suffix in text order

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

PINGO how fast can we compute $NSV/PSV$?
both arrays can be computed in linear time

- consider the PSV array
  - NSV works analogously
- prepend $-\infty$ at index 0

Function \texttt{ComputePSV}(SA \textit{with} $-\infty$):

1. for $i = 1, \ldots, n$ do
2.  \hspace{1em} $j = i - 1$
3.  \hspace{1em} while $j \geq 1$ and $SA[i] < SA[j]$ do
4.     \hspace{2em} $j = \text{PSV}[j]$
5.     \hspace{2em} $\text{PSV}[i] = j$
6.     return $\text{PSV}$

example on the board
Both arrays can be computed in linear time.

Consider the PSV array.

\[ \text{NSV works analogously} \]

Prepend \(-\infty\) at index 0.

---

**Function** `ComputePSV(SA with \(-\infty\))`:

1. for \(i = 1, \ldots, n\) do
2. \(j = i - 1\)
3. while \(j \geq 1\) and \(SA[i] < SA[j]\) do
4. \(j = PSV[j]\)
5. \(PSV[i] = j\)
6. return \(PSV\)

---

Follow already computed values.

Nothing in between can be PSV.

Compare each element at most twice.

Compute PSV and NSV in \(O(n)\) time.

Example on the board.
Recap: Range Minimum Queries

- for a range \([\ell..r]\), return position of smallest entry in an array in that range
- query time: \(O(1)\) using \(O(n)\) space
- can be used to compute the \(lcp\)-value of any two suffixes using the \(LCP\)-array

- use all arrays to find lexicographically closest suffixes
- that occur before current suffix in text order

<table>
<thead>
<tr>
<th>(T)</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>b</th>
<th>a</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SA)</td>
<td>13</td>
<td>12</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>11</td>
<td>2</td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>(PSV)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>(NSV)</td>
<td>2</td>
<td>3</td>
<td>$\infty$</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>$\infty$</td>
<td>10</td>
<td>11</td>
<td>$\infty$</td>
<td>13</td>
<td>$\infty$</td>
</tr>
<tr>
<td>(LCP)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
LZ77 Factorization using \(SA, ISA, LCP, NSV, PSV,\) and \(RMQs\):

```plaintext
Function LZ77\((SA, ISA, LCP, RMQ, PSV, NSV)\):

```pos = 1
2 while \(pos \leq n\) do
3     \(psv = SA[PSV[ISA[pos]]]\)
4     \(nsv = SA[NSV[ISA[pos]]]\)
5     \(\text{if } lcp(pos, psv + 1) > lcp(pos + 1, nsv) \text{ then}\)
6         \(\ell = lcp(pos, psv + 1) \text{ and } p = psv\)
7     \text{else}\)
8         \(\ell = lcp(pos + 1, nsv) \text{ and } p = nsv\)
9     \(\text{if } \ell = 0 \text{ then } p = pos\)
10    \text{new factor} \((\ell, p)\)
11    \(pos = pos + \max\{\ell, 1\}\)
```

bring your own example 📝
Lemma: LZ77 Running Time

The LZ77 factorization of a text of length $n$ can be computed in $O(n)$ time

Proof (Sketch)

- $SA, LCP, PSV, NSV, RMQ_{LCP}$ can be computed in $O(n)$ time
- for each text position only $O(1)$ time
Definition: LZ78 Factorization

Given a text $T$ of length $n$ over an alphabet $\Sigma$, the **LZ78 factorization** is

- a set of $z$ factors $f_1, f_2, \ldots, f_z \in \Sigma^+$, such that
- $T = f_1f_2\ldots f_z$, $f_0 = \epsilon$ 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$$
Lempel-Ziv 78 [ZL78]

Definition: LZ78 Factorization

Given a text $T$ of length $n$ over an alphabet $\Sigma$, the 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$$
Definition: LZ78 Factorization

Given a text $T$ of length $n$ over an alphabet $\Sigma$, the **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$$
Lempel-Ziv 78 [ZL78]

Definition: LZ78 Factorization

Given a text $T$ of length $n$ over an alphabet $\Sigma$, the **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

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

$T = abababbbaba$

- $f_1 = a$
- $f_2 = b$
Definition: LZ78 Factorization

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

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

$Lempel-Ziv 78 \ [ZL78]$
Definition: LZ78 Factorization

Given a text $T$ of length $n$ over an alphabet $\Sigma$, the **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

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

$T = abababbbaba$

- $f_1 = a$
- $f_2 = b$
- $f_3 = ab$
- $f_4 = abb$
Definition: LZ78 Factorization

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

Example:

$T = \text{abababbbaba}$

- $f_1 = \text{a}$
- $f_2 = \text{b}$
- $f_3 = \text{ab}$
- $f_4 = \text{abb}$
- $f_5 = \text{bb}$
Definition: LZ78 Factorization

Given a text \( T \) of length \( n \) over an alphabet \( \Sigma \), the **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

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

\( T = \text{abababbbaba} \)$

- \( f_1 = a \)
- \( f_2 = b \)
- \( f_3 = ab \)
- \( f_4 = abb \)
- \( f_5 = bb \)
- \( f_6 = aba \)
Lempel-Ziv 78 [ZL78]

Definition: LZ78 Factorization

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

$T = \text{abababbbabab}$

- $f_1 = a$
- $f_2 = b$
- $f_3 = ab$
- $f_4 = abb$
- $f_5 = bb$
- $f_6 = aba$
- $f_7 = $
Definition: LZ78 Factorization

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

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

Example:

$T = \text{abababbbaba}$

- $f_1 = a$
- $f_2 = b$
- $f_3 = ab$
- $f_4 = abb$
- $f_5 = bb$
- $f_6 = aba$
- $f_7 = \$

$T = \text{abababbbaba}$
LZ78 Factorization using a Dynamic Trie

- use dynamic trie to hold computed factors
- our fastest easy to use dynamic trie is?
LZ78 Factorization using a Dynamic Trie

- use dynamic trie to hold computed factors
- our fastest easy to use dynamic trie is?
- using arrays of fixed size
LZ78 Factorization using a Dynamic Trie

- use dynamic trie to hold computed factors
- our fastest easy to use dynamic trie is?
- using arrays of fixed size

\[ T = abababbbababa$ \]

- \( f_1 = a \)
- \( f_2 = b \)
- \( f_3 = ab \)
- \( f_4 = abb \)
- \( f_5 = bb \)
- \( f_6 = aba \)
- \( f_7 = $ \)
Lemma:
The LZ78 factorization of a text of length $n$ can be computed in $O(n)$ time.
Lemma:
The LZ78 factorization of a text of length $n$ can be computed in $O(n)$ time

Proof (Sketch)
- search each character of the text at most once (in the trie)
- insert each character of the text at most once (in the trie)
memory usage of the LZ78 factorization very high using arrays of fixed size does not help
consider only a sliding window of the text
only factors in the window are found
space/compression rate trade-off
used in practice
This Lecture
- different compression methods for texts
- entropy coding
- dictionary compression

Linear Time Construction

Conclusion and Outlook
Conclusion and Outlook

This Lecture

- different compression methods for texts
- entropy coding
- dictionary compression

- LZ77 and LZ78 have been generalize, improved, and combined: a lot!
- LZ77
  - LZSS, LZB, LZR, LZH, . . .
- LZ78
  - LZC, LZY, LZW, LZFG, LZW, LZA, . . .

Linear Time Construction

![Diagram showing connections between ST, SA, LZ, and LCP]
Conclusion and Outlook

This Lecture
- different compression methods for texts
- entropy coding
- dictionary compression

- LZ77 and LZ78 have been generalize, improved, and combined: a lot!
- LZ77
  - LZSS, LZB, LZR, LZH, . . .
- LZ78
  - LZC, LZY, LZW, LZFG, LZW, LZJ, . . .

Linear Time Construction

Next Lecture
- easy to compress index
Bibliography I


Bibliography II


