Karlsruhe Institute of Technology

Text Indexing

Lecture 04: Text-Compression

Florian Kurpicz

PINGO

https://pingo.scc.kit.edu/204437

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 $i \leq j \in [1..n]$

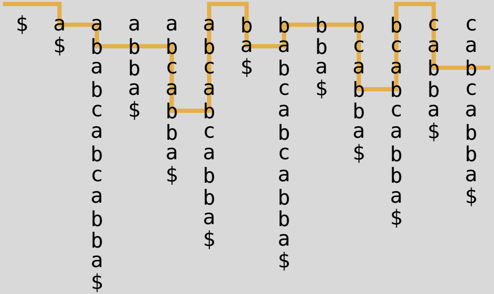
$$T[SA[i]..n] \leq T[SA[j]..n]$$

Definition: Longest Common Prefix Array

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

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

	1	2	3	4	5	6	7	8	9	10	11	12	13
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



Why Compression

Karlsruhe Institute of Technology

Types of Compression

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

Why Compression

Types of Compression

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

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

Why Compression

Karlsruhe Institute of Technology

Types of Compression

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

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

Types of Text-Compression

- entropy coding ℹ compress characters
- dictionary compression ℹ compress substings
- ...

Why Compression

Karlsruhe Institute of Technology

Types of Compression

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

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

Types of Text-Compression

- entropy coding ℹ compress characters
- dictionary compression ℹ compress substings
- . . .

This Lecture

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

k-th Order Empirical Entropy [KM99] (1/2)

Karlsruhe Institute of Technology

Definition: Histogram

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

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

k-th Order Empirical Entropy [KM99] (1/2)

Definition: Histogram

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

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

Definition: 0-th Order Empirical Entropy

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

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

k-th Order Empirical Entropy [KM99] (1/2)

Definition: Histogram

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

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

Definition: 0-th Order Empirical Entropy

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

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

- $T = \texttt{abbaaacaaba\$}$
- $n = 12$
- $Hist[\mathrm{a}] = 7$
- $Hist[\mathrm{b}] = 3$
- $Hist[\mathrm{c}] = 1$
- $Hist[\$] = 1$

k-th Order Empirical Entropy [KM99] (1/2)

Definition: Histogram

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

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

Definition: 0-th Order Empirical Entropy

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

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

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

k-th Order Empirical Entropy [KM99] (1/2)

Definition: Histogram

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

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

Definition: 0-th Order Empirical Entropy

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

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

- $T = \mathtt{abbaaacaaba\$}$
- $n = 12$
- $Hist[\mathtt{a}] = 7$
- $Hist[\mathtt{b}] = 3$
- $Hist[\mathtt{c}] = 1$
- $Hist[\mathtt{\$}] = 1$
- $H_0(T) = (1/12)(7 \lg(12/7) + 3 \lg(12/3) + 1 \lg(12/1) + 1 \lg(12/1)) \approx 1.55$

k-th Order Empirical Entropy (2/2)

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

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

Definition: k-th Order Empirical Entropy

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

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

Example for k-th Order Empirical Entropy [Kur20]

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

Example for k-th Order Empirical Entropy [Kur20]

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

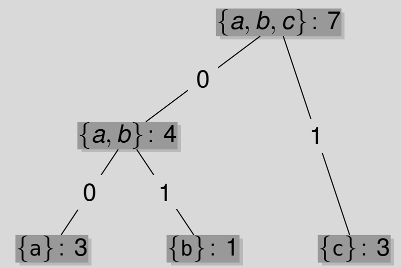
- does not measure repetitions well
- there are other measures 

Huffman Coding [Huf52]

- idea is to create a binary tree
- each character α is a leaf and has weight $\mathit{Hist}[\alpha]$
- 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

$T = \texttt{cbcacaa}$

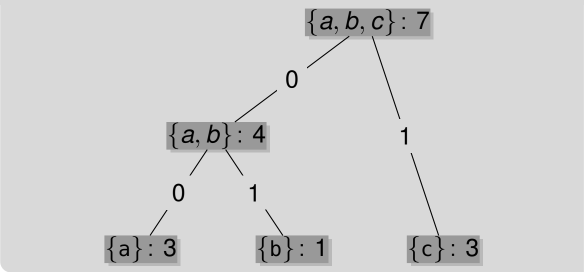


Huffman Coding [Huf52]

- idea is to create a binary tree
- each character α is a leaf and has weight $Hist[\alpha]$
- 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

$T = \texttt{cbcacaa}$



- codes are variable length and prefix-free
- tree/dictionary needed for decoding

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 $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ code for `c`
- add 1 and append 0
- 10 $\rightarrow$ code for `a`

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 $\rightarrow$ code for `c`
- add 1 and append 0
- 10 $\rightarrow$ 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 $\rightarrow$ code for `c`
- add 1 and append 0
- 10 $\rightarrow$ code for `a`
- add 1
- 11 $\rightarrow$ 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
  ℹ will be discussed in a later lecture

Continue From Last Slide

- length 1: `c`
- length 2: `a,b`
- start with 0 $\rightarrow$ code for `c`
- add 1 and append 0
- 10 $\rightarrow$ code for `a`
- add 1
- 11 $\rightarrow$ 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
  ℹ will be discussed in a later lecture

Continue From Last Slide

- length 1: `c`
- length 2: `a`,`b`
- start with 0 $\rightarrow$ code for `c`
- add 1 and append 0
- 10 $\rightarrow$ code for `a`
- add 1
- 11 $\rightarrow$ code for `b`

- still variable length and prefix-free
- instead of tree only require lengths' of codes and corresponding characters

Shannon-Fano Coding [Fan49; Sha48]

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

Shannon-Fano Coding [Fan49; Sha48]

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

- show that there always exists such a code
- assume a complete binary tree of depth $\ell_{\max} = \max_{\alpha \in \Sigma} \ell_\alpha$ 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]

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

- show that there always exists such a code
- assume a complete binary tree of depth $\ell_{\max} = \max_{\alpha \in \Sigma} \ell_\alpha$ 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

Proof there are enough free nodes (Sketch)

- a code ℓ_α marks $2^{\ell_{\max} - \ell_\alpha}$ nodes
- total number of marked leafs is

$$\begin{aligned}
\sum_{\alpha \in \Sigma} 2^{\ell_{\max} - \ell_\alpha} &= 2^{\ell_{\max}} \sum_{\alpha \in \Sigma} 2^{-\ell_\alpha} \\
&= 2^{\ell_{\max}} \sum_{\alpha \in \Sigma} 2^{-\lceil \lg \frac{n}{Hist[\alpha]} \rceil} \\
&\leq 2^{\ell_{\max}} \sum_{\alpha \in \Sigma} 2^{-\lg \frac{n}{Hist[\alpha]}} \\
&= 2^{\ell_{\max}} \sum_{\alpha \in \Sigma} \frac{Hist[\alpha]}{n} \\
&= 2^{\ell_{\max}}
\end{aligned}$$

Optimality of Both

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

Optimality of Both

Karlsruhe Institute of Technology

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

- one can show that no fixed-letter code can be better than Huffman not in this lecture
- 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
- $nH_o(T)$ is lower bound for compression without context

- one can show that no fixed-letter code can be better than Huffman ℹ not in this lecture
- Shannon-Fano codes can be slightly longer than Huffman
- even Shannon-Fano achieves H_0-compression

Proof

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

$$
\begin{aligned}
(1/n)|T_{SF}| &= (1/n)\sum_{\alpha\in\Sigma} Hist[\alpha]\lceil \lg \frac{n}{Hist[\alpha]}\rceil \\
&\leq \sum_{\alpha\in\Sigma} \frac{Hist[\alpha]}{n}(\lg \frac{n}{Hist[\alpha]} + 1) \\
&= \sum_{\alpha\in\Sigma} \frac{Hist[\alpha]}{n} \lg \frac{n}{Hist[\alpha]} + \sum_{\alpha\in\Sigma} \frac{Hist[\alpha]}{n} \\
&= H_0(T) + 1
\end{aligned}
$$

Problem with the Previous Approaches

Karlsruhe Institute of Technology

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

- does not work well with repetitions
- better encode $605 \times$ a

Lempel-Ziv 77 [ZL77]

Definition: LZ77 Factorization

Given a text T of length n over an alphabet Σ, 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 ≥ 2 times in $f_1 \ldots f_i$

Lempel-Ziv 77 [ZL77]

Definition: LZ77 Factorization

Given a text T of length n over an alphabet Σ, 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 ≥ 2 times in $f_1 \ldots f_i$

$T = \texttt{abababbbbaba\$}$

Lempel-Ziv 77 [ZL77]

Definition: LZ77 Factorization

Given a text T of length n over an alphabet Σ, 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 ≥ 2 times in $f_1 \ldots f_i$

$T = $ `ababababbbbaba$`

- $f_1 = $ a

Lempel-Ziv 77 [ZL77]

Definition: LZ77 Factorization

Given a text T of length n over an alphabet Σ, 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 ≥ 2 times in $f_1 \ldots f_i$

$T = $ `ababababbbbaba$`

- $f_1 = $ a
- $f_2 = $ b

Lempel-Ziv 77 [ZL77]

Definition: LZ77 Factorization

Given a text T of length n over an alphabet Σ, 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 ≥ 2 times in $f_1 \ldots f_i$

$T =$ `abababbbbaba$`

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ abab

Lempel-Ziv 77 [ZL77]

Definition: LZ77 Factorization

Given a text T of length n over an alphabet Σ, 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 ≥ 2 times in $f_1 \ldots f_i$

$T =$ `abababbbbaba$`

- $f_1 =$ `a`
- $f_2 =$ `b`
- $f_3 =$ `abab`
- $f_4 =$ `bbb`

Lempel-Ziv 77 [ZL77]

Definition: LZ77 Factorization

Given a text T of length n over an alphabet Σ, 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 ≥ 2 times in $f_1 \ldots f_i$

$T =$ `abababbbbaba$`

- $f_1 =$ `a`
- $f_2 =$ `b`
- $f_3 =$ `abab`
- $f_4 =$ `bbb`
- $f_5 =$ `aba`

Lempel-Ziv 77 [ZL77]

Definition: LZ77 Factorization

Given a text T of length n over an alphabet Σ, the **LZ77 factorization** is

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

$T =$ ababababbbbaba\$

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ abab
- $f_4 =$ bbb
- $f_5 =$ aba
- $f_6 =$ \$

Lempel-Ziv 77 [ZL77]

Definition: LZ77 Factorization

Given a text T of length n over an alphabet Σ, 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 ≥ 2 times in $f_1 \ldots f_i$

$T = \mathtt{abababbbbaba\$}$

- $f_1 = \mathtt{a}$
- $f_2 = \mathtt{b}$
- $f_3 = \mathtt{abab}$
- $f_4 = \mathtt{bbb}$
- $f_5 = \mathtt{aba}$
- $f_6 = \mathtt{\$}$

$T = \underbrace{\mathtt{aaa\ldots aa}}_{n-1 \text{ times}}\mathtt{\$}$

- $f_1 = \mathtt{a}$
- $f_2 = \underbrace{\mathtt{aaa\ldots aa}}_{n-2 \text{ times}}$
- $f_3 = \mathtt{\$}$

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-ℓ_i substring
  - $f_i = T[p_i..p_i + \ell_i)$

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-ℓ_i substring
  - $f_i = T[p_i..p_i + \ell_i)$

$T =$ ababab bbbaba$

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ abab
- $f_4 =$ bbb
- $f_5 =$ 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-ℓ_i substring
  - $f_i = T[p_i..p_i + \ell_i)$

$T = \mathtt{abababbbbaba\$}$

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

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-ℓ_i substring
  - $f_i = T[p_i..p_i + \ell_i)$

$T = \texttt{abababbbbaba\$}$

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

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-ℓ_i substring
  - $f_i = T[p_i..p_i + \ell_i)$

$T = \mathtt{ababab bbbaba\$}$

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

- finding the right-most reference is hard 

Previous and Next Smaller Values (1/2)

Definition: Previous and Next Smaller Value Arrays

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

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

	1	2	3	4	5	6	7	8	9	10	11	12	13
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
PSV	0	0	0	3	3	3	6	3	8	8	8	11	11
NSV	2	3	∞	5	6	8	8	∞	10	11	∞	13	∞
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3

Previous and Next Smaller Values (1/2)

Karlsruhe Institute of Technology

Definition: Previous and Next Smaller Value Arrays

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

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

In the Context of SA

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

	1	2	3	4	5	6	7	8	9	10	11	12	13
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
PSV	0	0	0	3	3	3	6	3	8	8	8	11	11
NSV	2	3	∞	5	6	8	8	∞	10	11	∞	13	∞
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3

Previous and Next Smaller Values (2/2)

- both arrays can be computed in linear time
- consider the PSV array
  NSV works analogously
- prepend $-\infty$ at index 0

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

> **for** $i = 1, \ldots, n$ **do**
> $j = i - 1$
>   **while** $j \geq 1$ and $SA[i] < SA[j]$ **do**
> $j = PSV[j]$
> $PSV[i] = j$
> **return** *PSV*

Previous and Next Smaller Values (2/2)

- both arrays can be computed in linear time
- consider the PSV array
  NSV works analogously
- prepend $-\infty$ at index 0

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

```
for i = 1, ..., n do
    j = i − 1
    while j ≥ 1 and SA[i] < SA[j] do
        j = PSV[j]
    PSV[i] = j
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

NSV, PSV, and RMQ

Karlsruhe Institute of Technology

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

	1	2	3	4	5	6	7	8	9	10	11	12	13
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
PSV	0	0	0	3	3	3	6	3	8	8	8	11	11
NSV	2	3	∞	5	6	8	8	∞	10	11	∞	13	∞
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3

LZ77 Factorization using *SA*, *ISA*, *LCP*, *NSV*, *PSV*, and *RMQ*s

Karlsruhe Institute of Technology

Function $\mathit{LZ77}(\mathit{SA}, \mathit{ISA}, \mathit{LCP}, \mathit{RMQ}, \mathit{PSV}, \mathit{NSV})$**:**

```
pos = 1
while pos ≤ n do
    psv = SA[PSV[ISA[pos]]]
    nsv = SA[NSV[ISA[pos]]]
    if lcp(pos, psv + 1) > lcp(pos + 1, nsv) then
        ℓ = lcp(pos, psv + 1) and p = psv
    else
        ℓ = lcp(pos + 1, nsv) and p = nsv
    if ℓ = 0 then p = pos
    new factor (ℓ, T[pos])
    pos = pos + max{ℓ, 1}
```

- bring your own example

LZ77: Running Time

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

Lempel-Ziv 78 [ZL78]

Definition: LZ78 Factorization

Given a text T of length n over an alphabet Σ, 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

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

Lempel-Ziv 78 [ZL78]

Definition: LZ78 Factorization

Given a text T of length n over an alphabet Σ, 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

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

$T = \texttt{abababbbbaba\$}$

Lempel-Ziv 78 [ZL78]

Definition: LZ78 Factorization

Given a text T of length n over an alphabet Σ, 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

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

$T = \texttt{ababab bbbaba\$}$

- $f_1 = \texttt{a}$

Lempel-Ziv 78 [ZL78]

Definition: LZ78 Factorization

Given a text T of length n over an alphabet Σ, 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

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

$T =$ `ababab bbbaba$`

- $f_1 =$ a
- $f_2 =$ b

Lempel-Ziv 78 [ZL78]

Definition: LZ78 Factorization

Given a text T of length n over an alphabet Σ, 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

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

$T =$ `ababab bbbaba$`

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ ab

Lempel-Ziv 78 [ZL78]

Definition: LZ78 Factorization

Given a text T of length n over an alphabet Σ, 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

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

$T =$ `ababab``bbbaba$`

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ ab
- $f_4 =$ abb

Lempel-Ziv 78 [ZL78]

Definition: LZ78 Factorization

Given a text T of length n over an alphabet Σ, 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

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

$T =$ `abababbbbaba$`

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ ab
- $f_4 =$ abb
- $f_5 =$ bb

Lempel-Ziv 78 [ZL78]

Definition: LZ78 Factorization

Given a text T of length n over an alphabet Σ, 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

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

$T = \mathtt{ababababbbbaba\$}$

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

Lempel-Ziv 78 [ZL78]

Definition: LZ78 Factorization

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

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

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

$T = \mathtt{abababbbbaba\$}$

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

Lempel-Ziv 78 [ZL78]

Definition: LZ78 Factorization

Given a text T of length n over an alphabet Σ, 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

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

$T = \mathtt{abababbbbaba\$}$

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

- $T = \mathtt{abababbbbaba\$}$

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

$T = \mathtt{ababababbbbaba\$}$

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

LZ78 Factorization in Linear Time

Lemma:

The LZ78 factorization of a text of length n can be computed in $O(n)$ time

LZ78 Factorization in Linear Time

Karlsruhe Institute of Technology

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)

Sliding Window

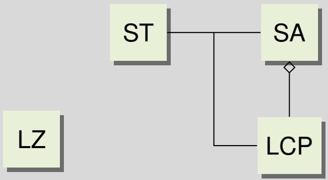
- memory usage of the LZ78 factorization very high ℹ using arrays of fixed size does not help
- consider only a sliding window of the text
- only factors in the window are found
- space/compression rate trade-off
- used in practice

Conclusion and Outlook

This Lecture

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

Linear Time Construction



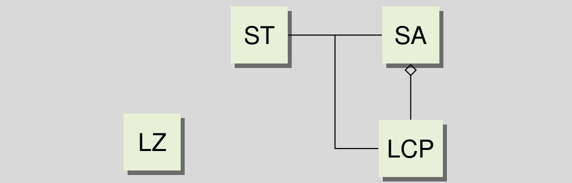
Conclusion and Outlook

This Lecture

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

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

Linear Time Construction



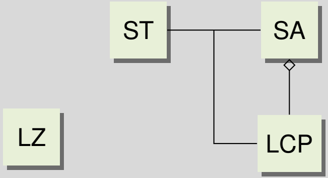
Conclusion and Outlook

This Lecture

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

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

Linear Time Construction



Next Lecture

- easy to compress index

One More Thing: 1/3-Evaluation

- finished $\approx 1/3$ of lectures
- short feedback round
- self evaluation
- what did you like
- what can be improved
- what is missing

Bibliography I

[Fan49] Robert M. Fano. *The Transmission of Information*. Massachusetts Institute of Technology, Research Laboratory of Electronics, 1949.

[GBS92] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. “New Indices for Text: Pat Trees and Pat Arrays”. In: *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992, pages 66–82.

[Huf52] David A. Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. In: *Proceedings of the IRE* 40.9 (1952), pages 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[KM99] S. Rao Kosaraju and Giovanni Manzini. “Compression of Low Entropy Strings with Lempel-Ziv Algorithms”. In: *SIAM J. Comput.* 29.3 (1999), pages 893–911. DOI: 10.1137/S0097539797331105.

[Kur20] Florian Kurpicz. “Parallel Text Index Construction”. PhD thesis. Technical University of Dortmund, Germany, 2020. DOI: 10.17877/DE290R-21114.

Bibliography II

[MM93] Udi Manber and Eugene W. Myers. "Suffix Arrays: A New Method for On-Line String Searches". In: *SIAM J. Comput.* 22.5 (1993), pages 935–948. DOI: 10.1137/0222058.

[Sha48] Claude E. Shannon. "A mathematical theory of communication". In: *Bell Syst. Tech. J.* 27.3 (1948), pages 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[ZL77] Jacob Ziv and Abraham Lempel. "A Universal Algorithm for Sequential Data Compression". In: *IEEE Trans. Inf. Theory* 23.3 (1977), pages 337–343. DOI: 10.1109/TIT.1977.1055714.

[ZL78] Jacob Ziv and Abraham Lempel. "Compression of Individual Sequences via Variable-Rate Coding". In: *IEEE Trans. Inf. Theory* 24.5 (1978), pages 530–536. DOI: 10.1109/TIT.1978.1055934.