ADS: Algorithmen und Datenstrukturen 2

Teil 2

Prof. Dr. Gerhard Heyer

Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Universität Leipzig

18. April 2018

[Letzte Aktualisierung: 07/05/2018, 19:45]

Die LZ-Familie von Kompressionsalgorithmen

LZ77, LZ78, LZW, ...

Nutze unterschiedliche Häufigkeit von *Zeichenfolgen* aus.

Grundlegende Idee: verwende ein "Wörterbuch" von Zeichenfolgen, so dass häufige Zeichenfolgen durch Verweise auf das Wörterbuch platzsparend codiert werden können. (LZ78,LZW: explizites Wörterbuch; LZ77: implizit, "sliding window")

Text-Beispiel: RABARBABARBARABARBARBARENBART[1]

Grundsätzlicher Konflikt: Je grösser das Wörterbuch, desto

+ mehr und längere Zeichenfolgen können durch Verweise codiert werden
- mehr Platzbedarf pro Verweis
- mehr Platzbedarf für Wörterbuch
- höhere Laufzeit und Platzbedarf für Kompression und Dekompression

[1] in kompressionsfreundlich-reformierter Rechtschreibung

Lempel-Ziv-Welch (LZW)

Algorithmus:

- Initialiere das Wörterbuch mit den erlaubten Zeichen.
- Kodierungsschleife:
  - Bestimme das längste Wort aus dem Wörterbuch, das mit dem Anfang des noch nicht kodierten Teils der Eingabefolge übereinstimmt.
  - Schreibe die Nummer dieses Worts in die Ausgabe.
  - Erstelle einen neuen Wörterbucheintrag: die Verlängerung der eben gefundenen Buchstabenfolge um den nächsten Buchstaben.
- Das Wörterbuch wird so immer größer. Deshalb: setze Maximalgrösse; bei Überschreiten initialisiere das Wörterbuch wieder neu.

LZW Algorithmus – Kodieren

Initialisiere Codetabelle (jedes Zeichen erhält einen Code/eine Nummer);
präfix := "";
while *Ende der Eingabe noch nicht erreicht* **do**
    suffix := nächstes Zeichen der Eingabe;
    wort := präfix + suffix;
    if *wort in Codetabelle* **then** präfix := wort;
     else
        gib Code von präfix aus;
        trage wort in Codetabelle ein;
        präfix := suffix;
    end
end
if $präfix \neq \emptyset$ **then**
    gib Code von präfix aus;
end

LZW Algorithmus – Beispiel "Kodieren"

Kodieren von ABCABCABCD

Initiale Codetabelle: 0:**A** 1:**B** 2:**C** 3:**D**

	präfix	wort	suffix	Eingabe	Code	Ausgabe
0		**A**	**A**	**ABCABCABCD**		
1	**A**	**AB**	**B**	**BCABCABCD**	4:**AB**	0 (**A**)
2	**B**	**BC**	**C**	**CABCABCD**	5:**BC**	1 (**B**)
3	**C**	**CA**	**A**	**ABCABCD**	6:**CA**	2 (**C**)
4	**A**	**AB**	**B**	**BCABCD**		
5	**AB**	**ABC**	**C**	**CABCD**	7:**ABC**	4 (**AB**)
6	**C**	**CA**	**A**	**ABCD**		
7	**CA**	**CAB**	**B**	**BCD**	8:**CAB**	6 (**CA**)
8	**B**	**BC**	**C**	**CD**		
9	**BC**	**BCD**	**D**	**D**	9:**BCD**	5 (**BC**)
10	**D**					3 (**D**)

LZW Algorithmus – Dekodieren

Initialisiere Codetabelle (jedes Zeichen erhält einen Code);
lies Code;
präfix := dekodiere Code (laut Codetabelle);
gib präfix aus;
while *Ende der Eingabe noch nicht erreicht* **do**
    lies Code;
    if *Code in Codetabelle* **then**
        wort := dekodiere Code (laut Codetabelle);
        neues_wort := präfix + erstes Zeichen von wort;
    else
        wort := präfix + erstes Zeichen von präfix;
        neues_wort := wort;
    end
    gib wort aus;
    trage neues_wort in Codetabelle ein;
    präfix := wort;
end

LZW Algorithmus – Beispiel "Dekodieren"

Dekodieren von 0 1 2 4 6 5 3

Initiale Codetabelle: 0:**A** 1:**B** 2:**C** 3:**D**

Code	präfix	wort	Code	Ausgabe
0	**A**			**A**
1	**A**	**B**	4:**AB**	**B**
2	**B**	**C**	5:**BC**	**C**
4	**C**	**AB**	6:**CA**	**AB**
6	**AB**	**CA**	7:**ABC**	**CA**
5	**CA**	**BC**	8:**CAB**	**BC**
3	**BC**	**D**	9:**BCD**	**D**

Zu Beachten: In Zeile '0' wird das Präfix ausgegeben, in allen folgenden Zeilen das Wort

Vorteile von LZW

- Wörterbuch und Codierung werden häufig gewechselt: LZW kann sich einem Kontextwechsel im Eingabestrom gut anpassen.
- Um den damit verbundenen “Gedächtnisverlust” zu beschränken, kann man die Zeichen aus dem Wörterbuch (nach Anzahl der Benutzung und Länge) bewerten und die besten n Zeichen behalten.

Lempel-Ziv Algorithmen

LZ77 (Sliding Window)

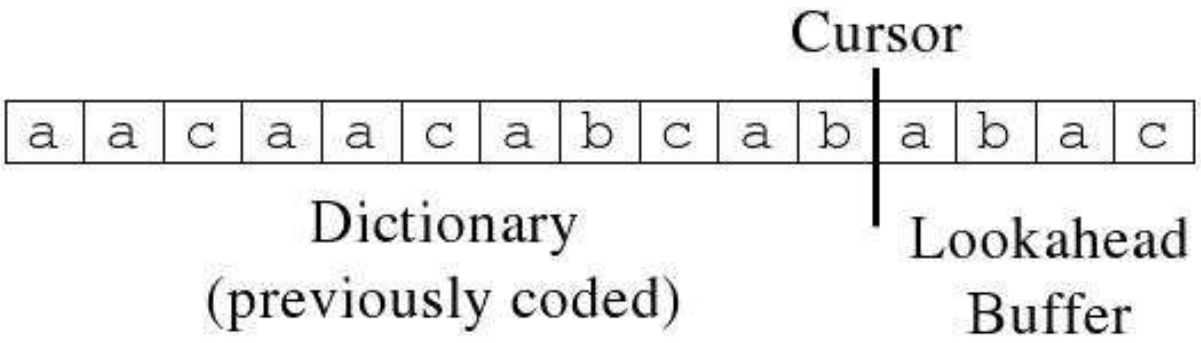
- Varianten: LZSS (Lempel-Ziv-Storer-Szymanski)
- Anwendungen: `gzip`, Squeeze, LHA, PKZIP, ZOO

LZ78 (explizites Dictionary)

- Varianten: LZW (Lempel-Ziv-Welch), LZC (Lempel-Ziv-Compress)
- Anwendungen: `compress`, GIF, CCITT (modems), ARC, PAK

LZ77 galt ursprünglich im Vergleich zu LZ78 als besser komprimierend, aber zu langsam; auf heutigen, leistungsfähigeren Rechnern ist LZ77 ausreichend schnell.

LZ77: Sliding Window Wörterbuch



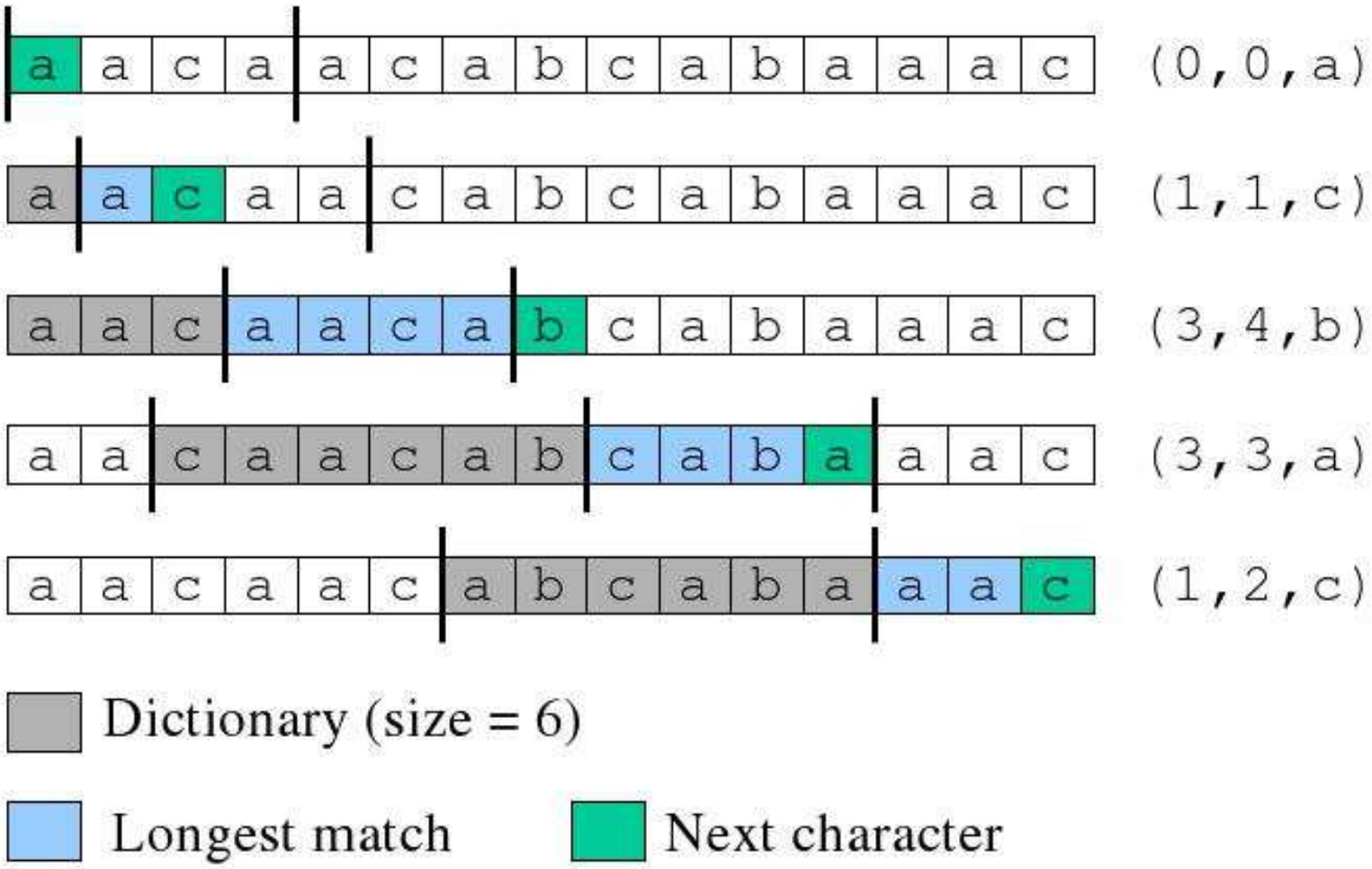
Dictionary- und Buffer-Windows haben feste Länge und verschieben sich zusammen mit dem Cursor.
An jeder Cursor-Position passiert folgendes:

- Ausgabe des Tripels `(p,l,c)`

  `p` = Beginne p Schritte links vom Cursor im Dictionary
  `l` = Länge des longest match
  `c` = nächstes Zeichen rechts vom longest match

- Verschiebe das Window um `l+1`

LZ77: Beispiel



LZ77 Dekodierung

Der Dekodierer arbeitet mit den selben Windowlängen wie der Kodierer

- Im Falle des Tripels `(p,l,c)` geht er `p` Schritte zurück, liest die nächsten `l` Zeichen und kopiert diese nach hinten. Dann wird noch `c` angefügt.

Was ist im Falle `l>p`? (d.h. nur ein Teil der zu kopierenden Nachricht ist im Dictionary)

- hier sei der lookahead-buffer beim Codieren 10 Zeichen groß gewesen
- Beispiel `dict = abcd, codeword = (2,9,e)`
- Lösung: Kopiere einfach zeichenweise:

```
for (i = 0; i < length; i++)
out[cursor+i] = out[cursor-offset+i]
```

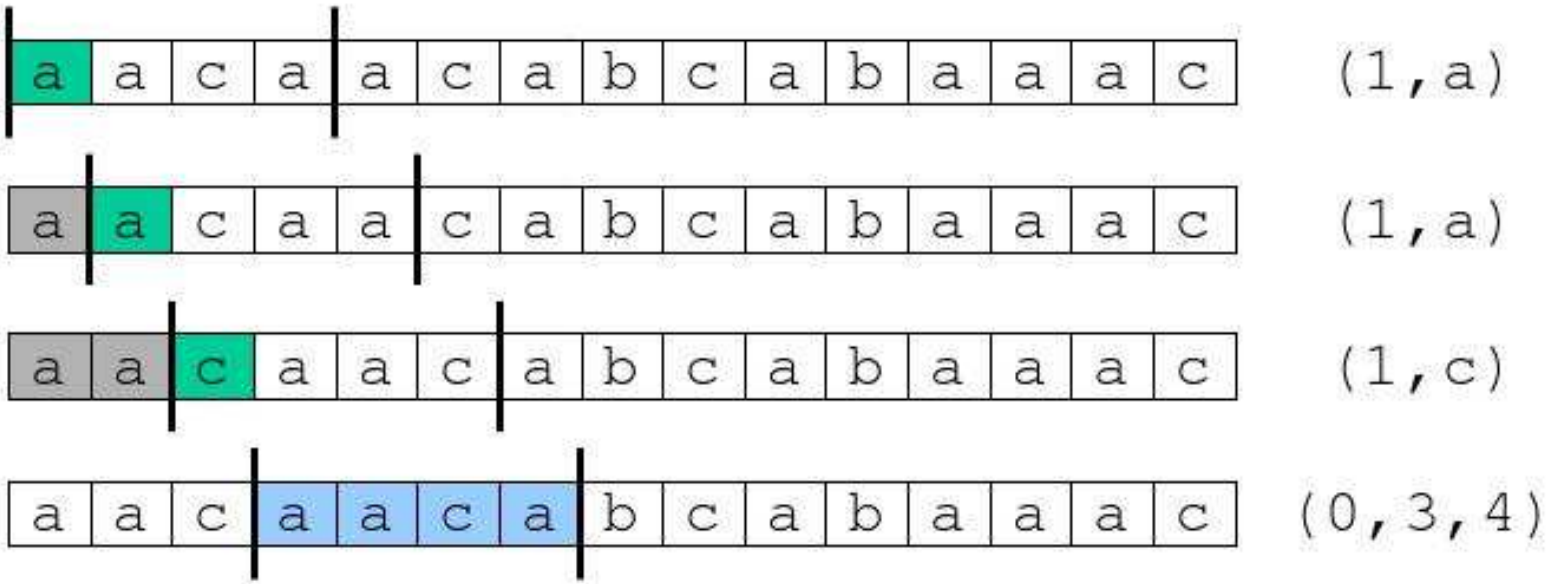
- `Out = abcdcdcdcdcdce`

Optimierungen in gzip: Verwendung von LZSS

LZSS verbessert LZ77: Im kodierten Output sind zwei verschiedene Formate erlaubt:

`(0, position, length)` oder `(1,char)`

Benutze das zweite Format, falls `length < 3`.



Weitere Optimierungen in `gzip`: `Deflate`-Algorithmus

- Nachträgliche Huffman-Codierung der Ausgabe
- Clevere Strategie bei der Codierung: Möglicherweise erlaubt ein kürzerer Match in aktuellen Schritt einen viel längeren Match im nächsten Schritt
- Benutze eine Hash-Tabelle für das Wörterbuch.
  - Hash-Funktion für Strings der Länge drei.
  - Suche für längere Strings im entsprechenden Überlaufbereich die längste Übereinstimmung.

Theorie zu LZ77

LZ77 ist *asymptotisch optimal* [Wyner-Ziv,94], d.h.
LZ77 komprimiert hinreichend lange Strings entsprechend seiner Entropie, falls die Fenstergröße gegen unendlich geht.

$$H_n = \sum_{X \in A^n} p(X) \log \frac{1}{p(X)}$$

$$H = \lim_{n \to \infty} H_n$$

Achtung: um nah an dieses Optimum zu kommen, braucht man sehr grosse Fenster. Als Default in gzip wird z.B. 32KB verwendet.

Kompression nach Burrows-Wheeler-Transformation

- Michael Burrows und David Wheeler, Mai 1994
- Verwendet in `bzip`
- Die Kompression ist sehr gut geeignet für Text, da Kontext berücksichtigt wird.
- **Grundidee:** Daten vorbehandeln, um sie besonders gut komprimieren zu können.
- Kompression in 3 Schritten
  - Burrows-Wheeler-Transformation (BWT):
    produziere lange Runs von Zeichen
  - Move-To-Front (MTF):
    produziere daraus lange runs von Nullen
  - dann erst komprimieren (Huffman-Codierung)

Beispiel für Burrows-Wheeler-Transformation

Vorwärtstransformation von `HelloCello`

	0	1	2	3	4	5	6	7	8	9
0	H	e	l	l	o	C	e	l	l	o
1	e	l	l	o	C	e	l	l	o	H
2	l	l	o	C	e	l	l	o	H	e
3	l	o	C	e	l	l	o	H	e	l
4	o	C	e	l	l	o	H	e	l	l
5	C	e	l	l	o	H	e	l	l	o
6	e	l	l	o	H	e	l	l	o	C
7	l	l	o	H	e	l	l	o	C	e
8	l	o	H	e	l	l	o	C	e	l
9	o	H	e	l	l	o	C	e	l	l

HelloCello

	0	1	2	3	4	5	6	7	8	9	
0	C	e	l	l	o	H	e	l	l	o	
1	H	e	l	l	o	C	e	l	l	o	*I*
2	e	l	l	o	C	e	l	l	o	H	
3	e	l	l	o	H	e	l	l	o	C	
4	l	l	o	C	e	l	l	o	H	e	
5	l	l	o	H	e	l	l	o	C	e	
6	l	o	C	e	l	l	o	H	e	l	
7	l	o	H	e	l	l	o	C	e	l	
8	o	C	e	l	l	o	H	e	l	l	
9	o	H	e	l	l	o	C	e	l	l	
										L	

ooHCeellll

Warum BWT?

- die Zeilen links über “HelloCello” sind zyklische Rotationen des Texts
- die Zeilen rechts sind lexikographisch sortiert
- Betrachten wir die 1. und letzte Spalte:
  - letzte Spalte: 1-Char Prefix der jeweiligen Zeile
  - ab 1. Spalte: Suffixe des jeweils letzten Buchstabens der Zeile
- für Suffixe mit häufig auftretendem Präfix tauchen die Präfix-Buchstaben gruppiert in der letzten Zeile auf
- (nicht immer perfekt gruppiert, aber in “guten” Runs)

- BWT hat eine inverse Funktion . . .

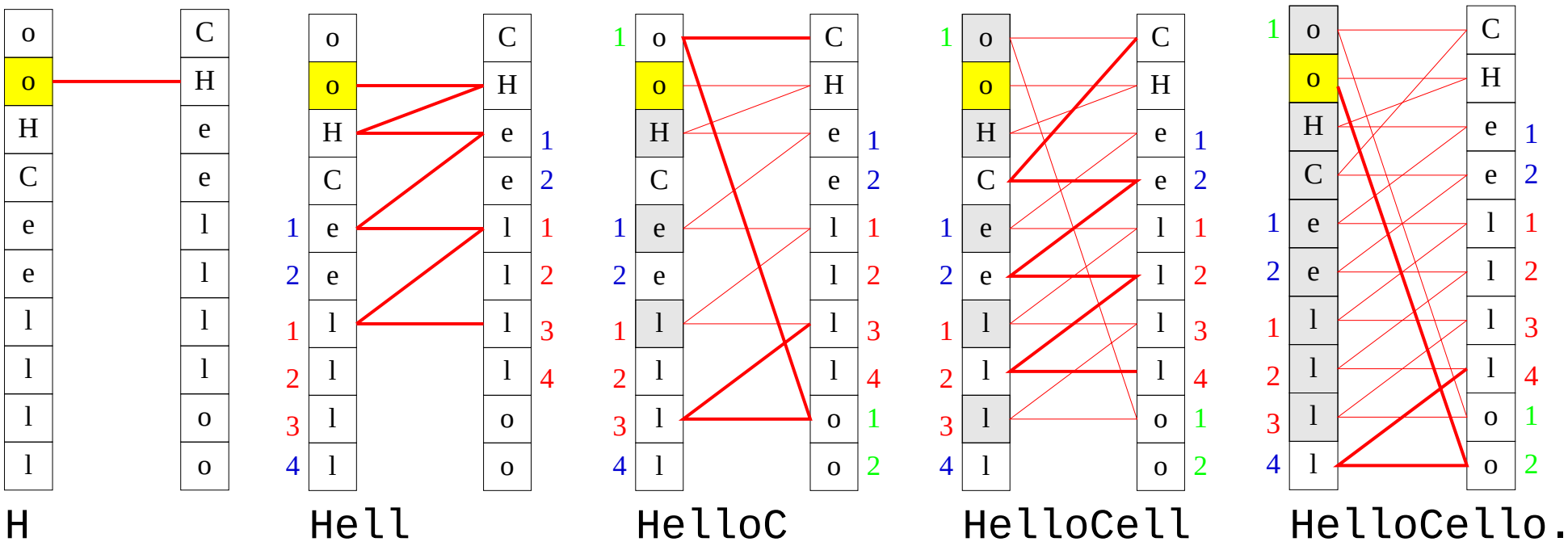
Rücktransformation

- Erstelle eine Zuordnung von Zeichen in codierter Position (letzte Spalte der Codierungstabelle) zur sortierten Position (erste Spalte der Codierungstabelle)
- Position in letzter Spalte entspricht Zeilennummer
- Position in erster Spalte übernimmt die Position aus der letzten Spalte
- Bestimme Startposition
  - Suche Zeile, welche uncodierten Text enthält
  - Startposition ist Zeilennummer in letzter Spalte
- Beginne bei Startposition
- While noch nicht alle Positionen besucht:
  - Lies Ergebnis aus sortierter Liste (*Zeile*3)
  - Gehe zu Position in Spalte codiert (*Zeile*4)

Zuordnung: Beispiel

codiert (letzte Spalte)	o	o	H	C	e	e	l	l	l	l
Position (letzte Spalte)	0	1	2	3	4	5	6	7	8	9
Sortiert (erste Spalte)	C	H	e	e	l	l	l	l	o	o
Position in Spalte codiert	3	2	4	5	6	7	8	9	0	1
Ausgabe		**H**	**e**		**l**		**l**		**o**	

Beispiel für BW Rücktransformation



Die Burrows-Wheeler Transformation

- Ein Eingabeblock der Länge N wird als quadratische Matrix dargestellt, die alle Rotationen des Eingabeblocks enthält.
- Die Zeilen der Matrix werden alphabetisch sortiert.
- Die letzte Spalte und die Zeilennummer des Originalblocks werden ausgegeben.

In der Ausgabe werden Zeichen mit ähnlichem Kontext zusammensortiert
$\Rightarrow$ lange Runs gleicher Buchstaben.

Daraus lässt sich der Originalblock wieder rekonstruieren
(wird hier nicht bewiesen, wurde nur demonstriert).

Kommentare zur Rücktransformation

- *Berechnung* der *Korrektheit* via equational Reasoning: “Functional Pearls: Inverting the Burrows-Wheeler Transform” (Bird, Mu, 2004)
- wichtig ist folgende Eigenschaft eines:
  - Sortieralgorithmus auf
  - Matrix von Rotationen eines Strings
- die Matrix nach Zeilen sortiert is äquivalent zu
- *n*-mal die komplette Matrix 1-Char rechts rotieren und *nur nach dem 1. Char* stabil zu sortieren
- mit diesem Argument kann die ursprüngliche sortierte Matrix (aus der Vorwärts-Variante) aus der letzten Spalte allein wieder hergestellt werden

Anmerkungen

- Gruppierung typischerweise nicht perfekt (im Vergleich zum Sortieren)
- aber reversibel (sonst wäre BWT fuer Kompressionsverfahren nicht nützlich)
- Eingaben sollten relativ lang sein, mehrere Kilobyte, um genügend lange Runs zu erzeugen
- BWT erfordert Prefix-Suffix Muster

MTF: Move-To-Front-Coding

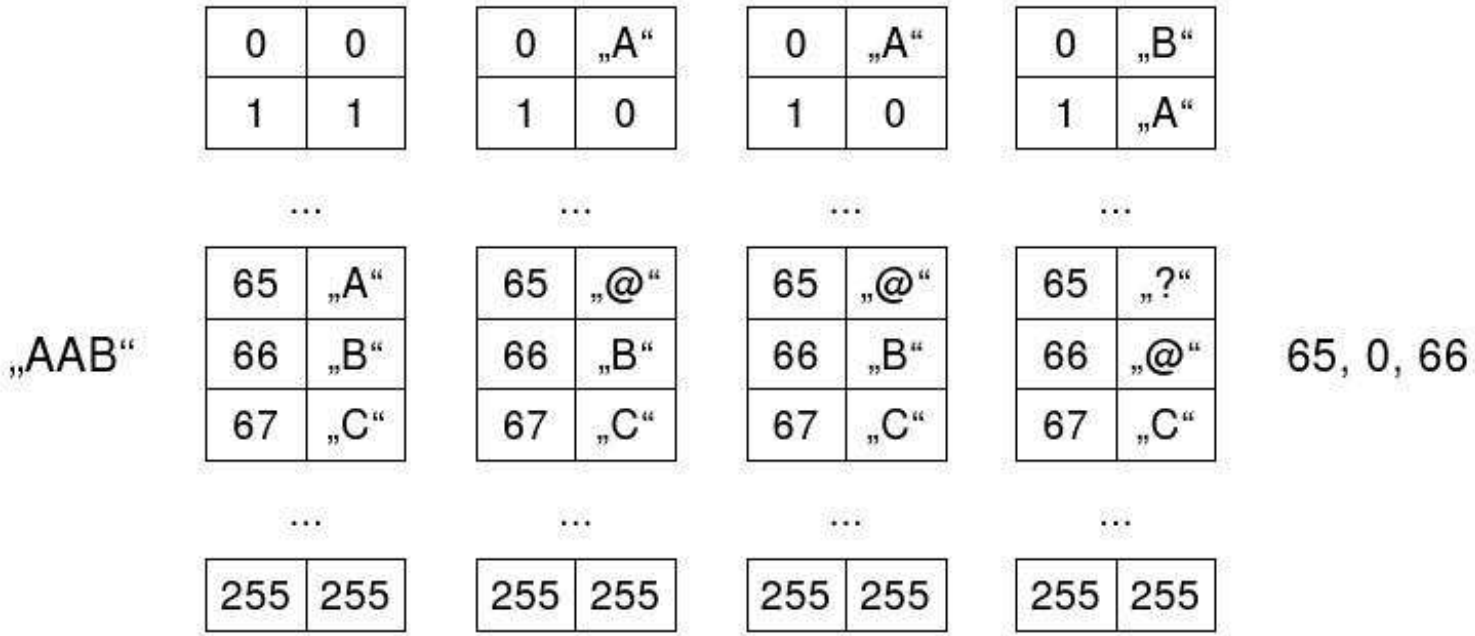
Idee: Codiere Runs von Buchstaben mit möglichst kleinen Zahlen

- Beginne mit Array `...A ...Z ...`, wobei 'A' = 65, 'B' = 66, ...
- das 1. Auftreten eines Buchstaben im Run codiert den (momentanen) Index des Buchstabens
- dieser Buchstabe wandert an den Anfang des Array
- weiteres Auftreten des Buchstaben wird durch '0' codiert

also:

1. Lese Buchstabe
2. finde Index zu Buchstabe und schreibe Index
3. Verschiebe Buchstabe an 0'te Position
4. Weiter mit Schritt 1, falls Eingabe nicht zu Ende

MTF: Move-To-Front-Coding



Eingabe:	A	A	B	B	C	C	C	A	A	B	C	C	C
Ausgabe:	65	0	66	0	67	0	0	2	0	2	2	0	0

MTF Decoding

1. Beginne mit Array `...A ...Z ...`, wobei 'A' = 65, 'B' = 66, ...
2. die Eingabe ist eine Liste der Indices
3. Schreibe Buchstaben welcher durch Index (zB. 65 $\rightarrow$ 'A') codiert
4. Verschiebe Buchstaben an den Anfang des Arrays
5. weiter mit Schritt 2

Kompression mit BWT – Zusammenfassung

bzip-Funktionen:

$$\textbf{BWT-Kompression}(x) = \text{Huffmann}(\text{MTF}(\text{BWT}(x)))$$

$$\textbf{BWT-Dekompression}(y) = \text{BTW}^{-1}(\text{MTF}^{-1}(\text{Huffmann}^{-1}(y)))$$

$$[\ = \text{BWT-Kompression}^{-1}(y)\ ]$$

Wozu das Ganze

- BWT erzeugt durch Sortierung lange Läufe gleicher Zeichen
- MTF erzeugt daraus kleine (Index-)Zahlen für häufig wiederholte Zeichen, genauer: Lange Folgen von '0'
- Transformation erlaubt gute Kompression:
  - Lauflängen-Kodierer (evtl. speziell für '0') oder
  - Huffman-Kodierer (möglichst kurze Bitfolge für häufige Zeichen)

Information, Entropie, Redundanz und Kompression

Definition (Shannon)

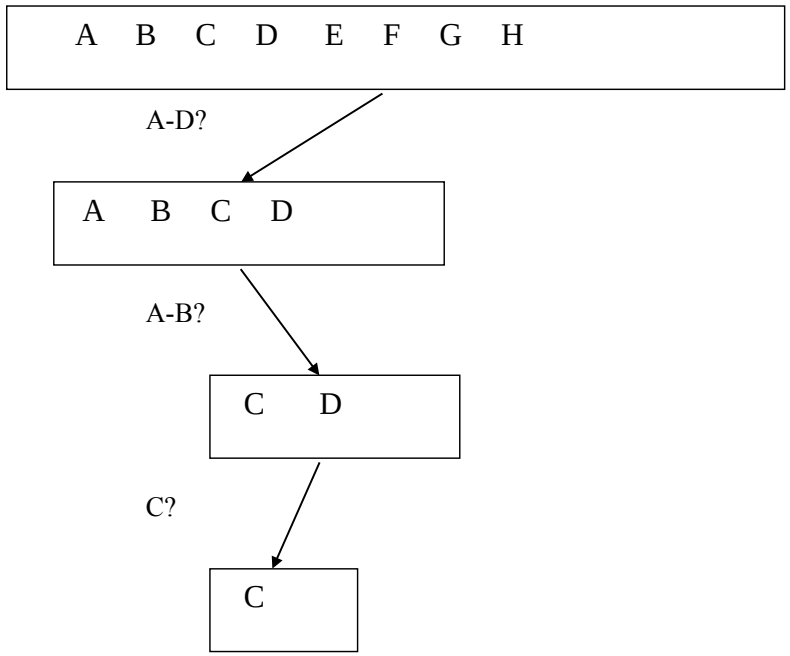
Entscheidungsinformation: Anzahl optimal gewählter binärer Entscheidungen zur Ermittlung eines Zeichens in einem Zeichenvorrat
Entropie: der mittlere Informationsgehalt eines Textes
Redundanz: Anteil einer Nachricht, der keine Information enthält

- gegeben sei ein Alphabet $\mathcal{A} = \{A \ldots Z\}$ (oder ein beliebiger anderer Zeichenvorrat)
- für jeden Buchstaben $x \in \mathcal{A}$ sei die Wahrscheinlichkeit, dass dieser Buchstabe auftritt: $p(x)$ mit $\sum_x p(x) = 1$.
- der Informationsgehalt sei $\mathtt{I}(x) = -\log_2 p(x)$
- hierbei meint *Information* im engeren Sinne *Entscheidungsinformation*, also die Anzahl optimal gewählter binärer Entscheidungen zur Ermittlung eines Zeichens innerhalb eines Zeichenvorrats

Entscheidungsinformation - Beispiel

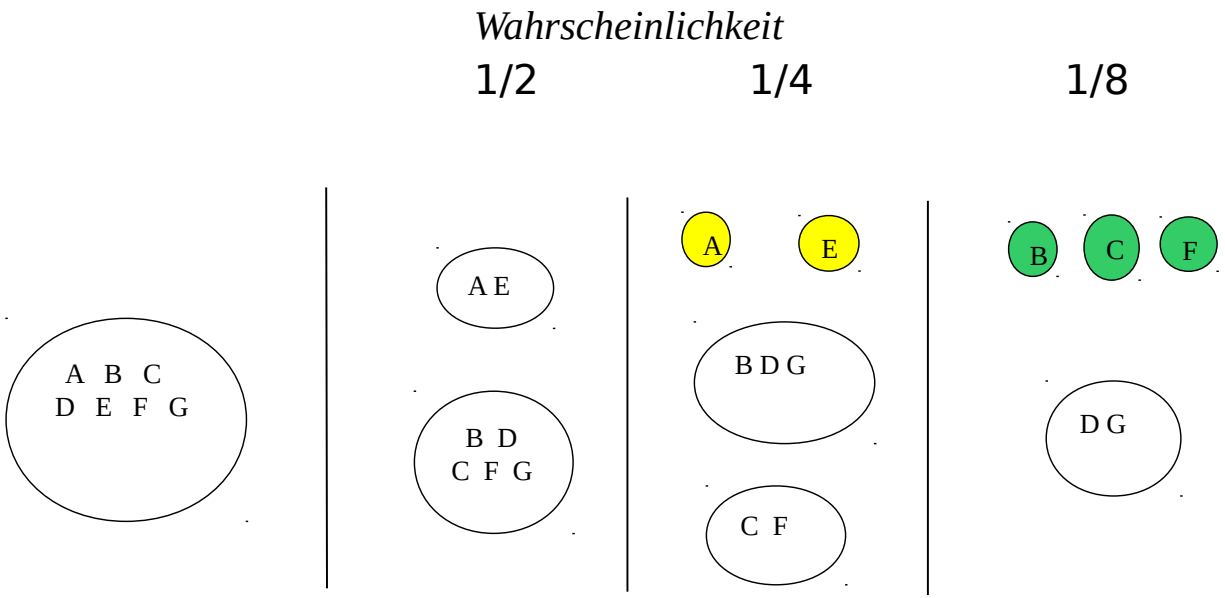
Gegeben seien 8 Zeichen. Nach maximal wieviel Schritten ist ein Zeichen gefunden?

Entscheidungsbaum:



Entscheidungsinformation - Beispiel

Wichtig für den allgemeinen Fall ist die Aufteilung eines Alphabets nicht in gleich große, sondern **gleich wahrscheinliche** Mengen von Zeichen.
Das i-te Zeichen ist nach k_i Alternativentscheidungen isoliert,
seine Wahrscheinlichkeit ist $p_i = (1/2)^k i$,
sein Informationsgehalt $k_i = \mathrm{ld}\ (1/p_i)$ bit.



Optimale Codierung

Buchstaben	p_i	**Codierung**
A	1/4	00
E	1/4	01
F	1/8	100
C	1/8	101
B	1/8	110
D	1/16	1110
G	1/16	1111

Mittlerer Entscheidungsgehalt pro Zeichen (Entropie):

$$H = p_1 l_1 + p_2 l_2 + ... + p_n l_n$$
$$= \sum p_i ld(1/p_i) bit$$
$$= 2/4 + 2/4 + 3/8 ... = 2{,}625$$

Entropie

- die *Entropie* eines Zeichens im Text ist:
  $H_1 = \sum_x p(x)\mathtt{I}(x) = -\sum_x p(x)\log_2 p(x)$
- für Wörter der Länge n ergibt sich:
  $H_n = -\sum_{w\in\mathcal{A}^n} p(w)\log_2 p(w)$
- mit $H = \lim_{n\to\infty} H_n/n$
- falls die Zeichen stoch. unabhängig sind: $H_n = nH_1$, also $H = H_1$

Redundanz

- besitzt in einer Codierung einer Nachricht das i-te Zeichen die Wortlänge N_i, so ist $L = \sum p_i N_i$ die **mittlere Wortlänge**.
- sind alle Elemente des Zeichenvorrats genau gleichwahrscheinlich, gilt $L = H$
- im allgemeinen gilt das *Shannonsche Codierungstheorem*
  - $H \leqq L$
  - jede Nachricht kann so codiert werden, dass die Differenz L-H beliebig klein wird
- Die Differenz L-H heißt *Code-Redundanz*, die Größe 1-H/L *relative Code-Redundanz*

Redundanz - Beispiel

Informationsgehalt Schriftsprache Deutsch

- 30 Buchstaben (inkl. Zwischenraum), also I = ld30 = 4,9 bit
- mittlerer Informationsgehalt *(Entropie)* unter Berücksichtigung von Bigrammen H = 1,6 bit
- **Redundanz:** 4,9 - 1,6 bit = 3,3 bit
- Text ist auch dann noch lesbar, wenn jeder zweite Buchstabe fehlt

Redundanz - konkret

Bei reduzierter Redundanz wird das Lesen sehr viel mühsamer

BEI REDUZIERTER REDUNDANZ WIRD DAS LESEN SEHR VIEL MÜHSAMER

BEIREDUZIERTERREDUNDANZWIRDDASLESENSEHRVIELMÜHSAMER

BE RE UZ ER ER ED ND NZ IR DA LE EN EH VI LM HS ME

(nach Breuer)

... und noch mal Kompression

Was bedeutet das alles?

- "zufällige" Texte haben die höchste Entropie und lassen sich nicht komprimieren
- falls häufig gleiche Zeichen hintereinander stehen, so sind diese Zeichen im Text stochastisch voneinander abhängig
- damit reduziert sich ihre Entropie und auch die des gesamten Textes
- Kompressionsverfahren nutzen dies, um möglichst nahe an das theoretische Optimum für Kompression zu kommen
- warum nicht "optimal": die Kompressionverfahren müssen die Abhängigkeiten sehen können