ADS: Algorithmen und Datenstrukturen 2

Teil 1

Prof. Dr. Gerhard Heyer

Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Universität Leipzig

3. April 2019

[Letzte Aktualisierung: 03/04/2019, 08:03]

Überblick

Einführung in allgemeine *Entwurfsprinzipien* für Algorithmen, die sich für viele Problemstellungen anwenden lassen.

- Greedy-Algorithmen
- Dynamische Programmierung
- Probabilistische und heuristische Algorithmen (für NP-schwierige Probleme)
- Maschinelles Lernen

Anwendungsschwerpunkte

- Datenkompression
- Graphen und Netzwerke
- Rucksackproblen, Travelling Salesman, ...

Datenkompression

Zwei Ziele bei der Formulierung von Algorithmen

- Effizienz bezüglich *Zeit* ODER
- Effizienz bezüglich *Speicherplatz*

Im folgenden:

- Datenkompression mit Hilfe effizienter *Kodierung*
- für eine *Speicher*-effiziente Darstellung

Datenkompression

Zwei grundsätzlich unterschiedliche Typen der Kompression

- **verlustfrei** (lossless)
  Codierung eindeutig umkehrbar.
- **verlustbehaftet** (lossy)
  Originaldaten können i.A. nicht eindeutig aus der komprimierten Codierung zurückgewonnen worden.
  Beispiele: JPEG, MPEG (mp3)

Kompressionsverfahren: Übersicht

- **Lauflängenkodierung:** Aufeinanderfolgende identische Zeichen werden zusammengefasst.
- **LZW, LZ77, gzip:** Mehrfach auftretende Zeichen*folgen* werden indiziert (*Wörterbuch*/*Codetabelle*), um dann durch ein einzelnes Zeichen dargestellt zu werden.
- **Huffman-Kodierung:** Häufiger auftretende Zeichen werden mit weniger Bits codiert.
- **Burrows-Wheeler:** Zeichenketten werden in besonders gut komprimierbare Zeichenketten vorverarbeitet.

Allgemeines Prinzip:
Ausnutzen von **Redundanz**
(Abweichungen von zufälliger Zeichenfolge, wiederkehrende Muster)

Lauflängencodierung

- Einfachster Typ von Redundanz:
  Läufe (runs) = lange Folgen sich wiederholender Zeichen.
- Beispiel:
  AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD.

Lauflängencodierung

- Einfachster Typ von Redundanz:
  Läufe (runs) = lange Folgen sich wiederholender Zeichen.
- Beispiel:
  AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD.
- Ersetze jeden Run durch seine Länge und das wiederholte Zeichen:
  4A3B2A5B8CDABCB3A4B3CD
- Lohnt sich nur für Läufe mit Länge > 2.

Wie können wir Text-Zeichen von Längenangaben unterschieden? Insbesondere, wenn alle Zeichen (auch Ziffern) im zu codierenden Text vorkommen dürfen.

Lauflängencodierung: Codierung mit Escape-Zeichen

- Wähle als **Escape-Zeichen** ein Zeichen Q, das in der Eingabe wahrscheinlich nur selten auftritt.
- Jedes Auftreten dieses Zeichens besagt, dass die folgenden beiden Buchstaben ein Paar (Zähler, Zeichen) bilden.
- Ein solches Tripel wird als *Escape-Sequenz* bezeichnet.
- Zählerwert i wird durch i-tes Zeichen des Alphabets dargestellt.
- Codierung erst für Läufe ab Länge 4 sinnvoll.
- Auftreten des Escape-Zeichens Q selbst muss speziell codiert werden, z.B. als Run der Länge 1 oder Q␣ (␣=Leerzeichen), etc.
- Beispiel:

  Text AAAABBBAABBBBBCCCCCCCCDABCQC.

Lauflängencodierung: Codierung mit Escape-Zeichen

- Wähle als **Escape-Zeichen** ein Zeichen Q, das in der Eingabe wahrscheinlich nur selten auftritt.
- Jedes Auftreten dieses Zeichens besagt, dass die folgenden beiden Buchstaben ein Paar (Zähler, Zeichen) bilden.
- Ein solches Tripel wird als *Escape-Sequenz* bezeichnet.
- Zählerwert i wird durch i-tes Zeichen des Alphabets dargestellt.
- Codierung erst für Läufe ab Länge 4 sinnvoll.
- Auftreten des Escape-Zeichens Q selbst muss speziell codiert werden, z.B. als Run der Länge 1 oder Q␣ (␣=Leerzeichen), etc.
- Beispiel:

Text	AAAABBBAABBBBBCCCCCCCCDABCQC.
Codierung	**Q**DABBBAA**Q**EB**Q**HCDABC**Q**␣C

Lauflängencodierung: Spezialfall "Binäre Daten"

Für Sequenzen von Bitwerten $\in \{0, 1\}$:

- Beobachtung: Läufe von 0 und 1 wechseln sich ab.
- Nutze das aus: gebe nur noch Lauflängen an.
- Beispiel:

Sequenz	000111011110110011111
Kodierung	3 3 1 4 1 2 3 5

Was ist, wenn die Sequenz mit 1 beginnt?

Codierung mit variabler Länge

- Ansatzpunkt für Kompression: In Sprache kommen Zeichen mit unterschiedlicher Häufgkeit vor, z.B. **E** häufiger als **Y**.
- Beispiel: ABRACADABRA
- Standardcodierung (unkomprimiert) mit 5 Bits pro Zeichen:
  00001 00010 10010 00001 00011 00001 00100 00001 00010 10010 00001
- Häufgkeit der Buchstaben:

A	B	C	D	R
5	2	1	1	2

- **Idee:** verwende für häufige Zeichen kurze Bitsequenzen, für seltene dafür längere. Minimiere so die Gesamtzahl der benötigten Bits.

Wie kann man, trotz unterschiedlich langer Bitfolgen, erkennen welche Bits welches Zeichen codieren?

Problem: Mehrdeutigkeit

Gegeben sei folgende Codierung mit 0 und 1 (bzw. kurz oder lang wie im Morsealphabet):

$E = 0$

$T = 1$

$A = 01$

Was bedeutet 0101?

Buchstabenkombination ETA AA ETET AET

Ohne Sonderzeichen keine eindeutige Zerlegung!

Codierung mit variabler Länge: Präfixcodes

Präfixcode (Präfix-freier Code)

- Kein Codewort ist Präfix eines anderen Codeworts.
- Durch Präfix-Freiheit: Codierung eindeutig umkehrbar.
- Im Beispiel (ABRACADABRA):

Buchstabe	A	B	C	D	R
Häufigkeit	5	2	1	1	2
Code	0	100	1010	1011	11

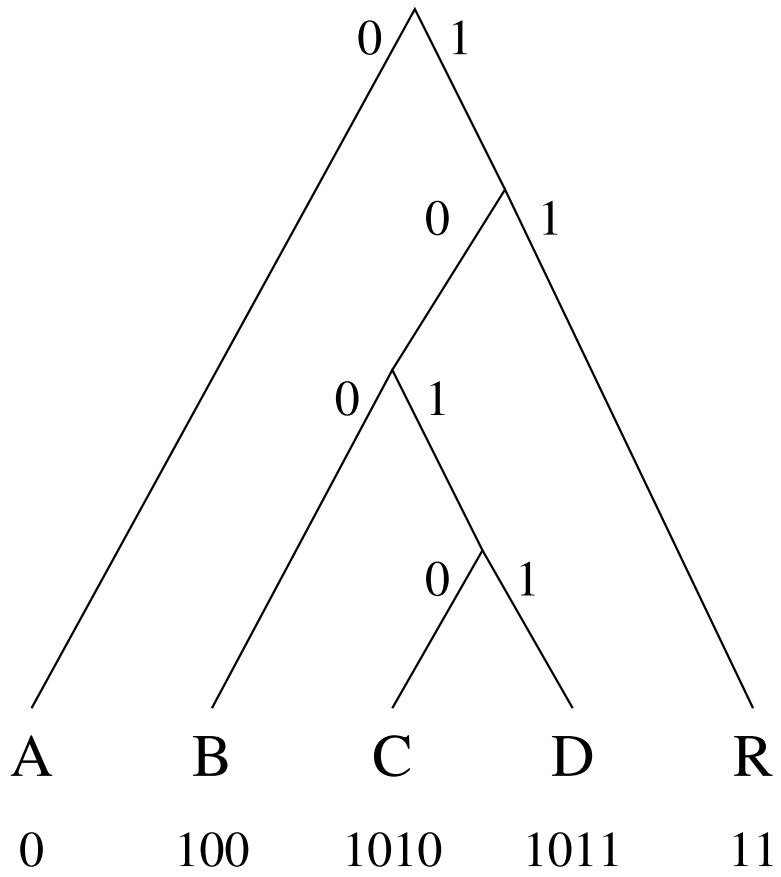
Codierung:

```
01001101010010110100110
AB  R AC   AD   AB  R A
```

Codierung mit variabler Länge: Tries

Binärer Präfixcode lässt sich durch Binärbaum (Trie) darstellen:

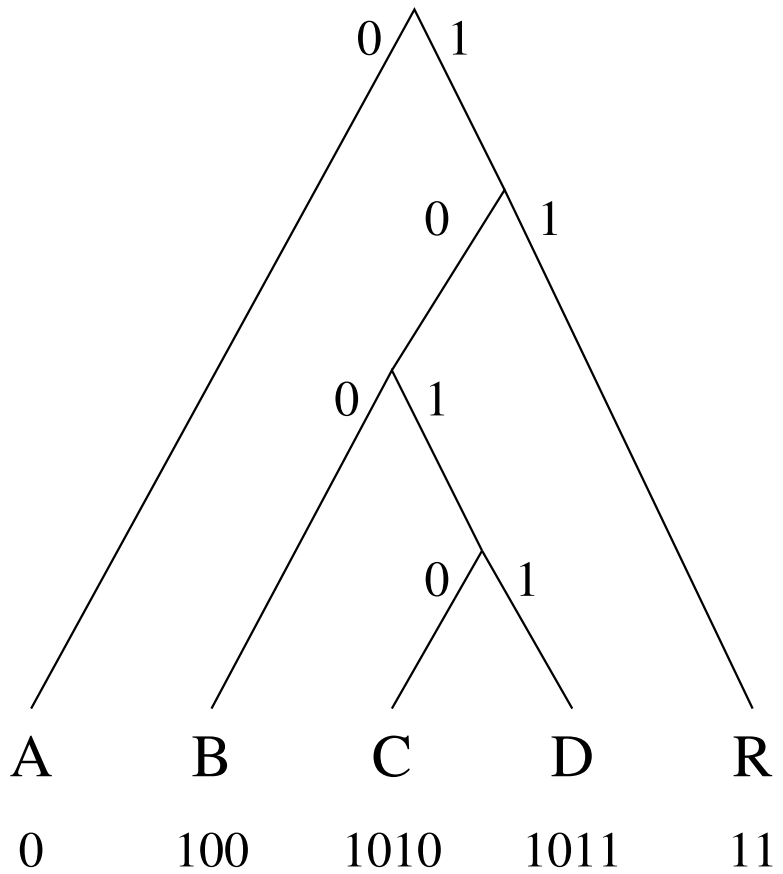
- **Blätter** = zu codierende Zeichen
- Zur **Codierung eines Zeichens**: laufe von Wurzel zum Blatt des Zeichens und gebe Kantenlabel aus (0 bei Verzweigung nach links; 1, nach rechts)
- übliche Bezeichnung: **Trie** (dazu später mehr)



Dekodierung (top-down)

Gegeben: Bitsequenz und Präfixcode als Trie

1. Beginne bei Wurzel im Baum
2. Wiederhole bis zum Ende der Bitsequenz
   1. Lies nächstes Bit. Bei 0 verzweige nach links im Baum, sonst nach rechts.
   2. Falls erreichter Knoten ein Blatt ist, gib das Zeichen des Blatts aus und springe zurück zur Wurzel.



Huffman-Code

Der *Huffman-Code* ist ein präfix-freier Code mit variabler Länge, der die unterschiedlichen Zeichen-Häufigkeiten ausnutzt und systematisch erzeugt werden kann.

Beispiel

"A SIMPLE STRING TO BE ENCODED USING
A MINIMAL NUMBER OF BITS"

Dazugehörige Häufigkeits-Tabelle

	_	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
count[k]	11	3	3	1	2	5	1	2	0	6	0	0	2	4	5	3	1	0	2	4	3	2	0	0	0	0	0

Der Huffman-Code

Erzeugung des Huffman-Codes

Schritt 1: Zähle Häufigkeit der Zeichen in der zu codierenden Zeichenfolge

Das folgende Programm ermittelt die Buchstaben-Häufigkeiten einer Zeichenfolge a und trägt diese in ein Feld `count` ein.

```
for ( i = 0 ; i <= 26 ; i++ ) count[i] = 0;
for ( i = 0 ; i < a.length ; i++ ) count[index(a[i])]++;
```

Dabei liefert `index(c)` den Index eines Zeichens c:
`index`(A)=1, `index`(B)=2, ...; `index`(␣):=0

Huffman-Code: Erzeugung des Tries

Schritt 2: Aufbau des Tries (entsprechend den Häufigkeiten)

Für jedes Zeichen mit Häufigkeit$\neq 0$, erzeuge einen Baum jeweils mit dem Zeichen als einzigem Knoten.

Schritt 3: Kombiniere iterativ die Bäume mit kleinsten Häufigkeiten

- Wähle zwei Bäume mit minimalen Häufigkeiten (Die Häufigkeit eines Baums ist die Summe der Häufigkeiten aller seiner Zeichen. Während der Erzeugung speichern wir Häufigkeit jedes Baums ab.)
- Verwende hierfür als Zwischenstruktur einen *MinHeap*
- Erzeuge neuen Knoten, der diese beiden Bäume als Nachfolger hat
- Iteriere Schritt 3 bis alle Knoten miteinander zu einem einzigen Baum verbunden sind.

Am Ende sind Zeichen mit geringen Häufigkeiten weit unten im Baum; Zeichen mit großen Häufigkeiten nah an der Wurzel.

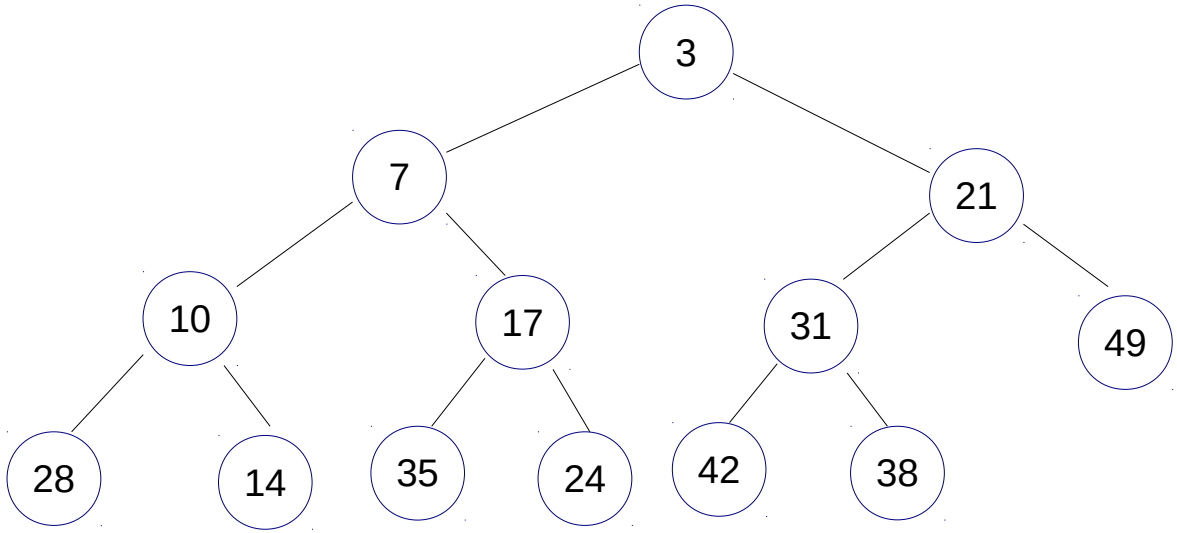
Kurze Erinnerung: Heap

- Datenstruktur Heap („Halde“) als Spezialfall eines binären Baumes
- **Ein Heap ist ein Binärbaum**, der
  - die Heapeingenschaft hat (Kinder sind größer/kleiner als der Vater)
  - bis auf die letzte Ebene vollständig besetzt ist
  - höchstens eine Lücke in der letzten Ebene hat, die ganz rechts liegt
- Heaps i. A. genutzt, um den Datentyp *Vorrangwarteschlange* (engl. „priority queue“) zu implementieren
- Folgende Operationen sollen dabei unterstützt werden:
  - Lesen des Objektes mit der kleinsten/größten Priorität
  - Löschen des Objektes mit der kleinsten/größten Priorität
  - Einfügen eines neuen Elements mit beliebiger Priorität

Anmerkungen

Aus der Definition folgt, dass

- ein Teilbaum eines Heaps wiederum ein Heap ist
- in einem beliebigen Pfad im Heap die Elemente sortiert sind
- Je nach dem, ob bei der Sortierung die Eigenschaft *größer* oder *kleiner* betrachtet wird, spricht man von *minHeap* oder *maxHeap*



Priority Queues

Typische Operationen auf Priority Queues:

- isempty, insert, readmin, delmin

Für das **Einfügen:**

- hänge das einzufügende Element an den freien Knoten der vorletzten Ebene
- stelle die Heap-Eigenschaft wieder her

Für das **Entnehmen** eines minimalen/maximalen Elements:

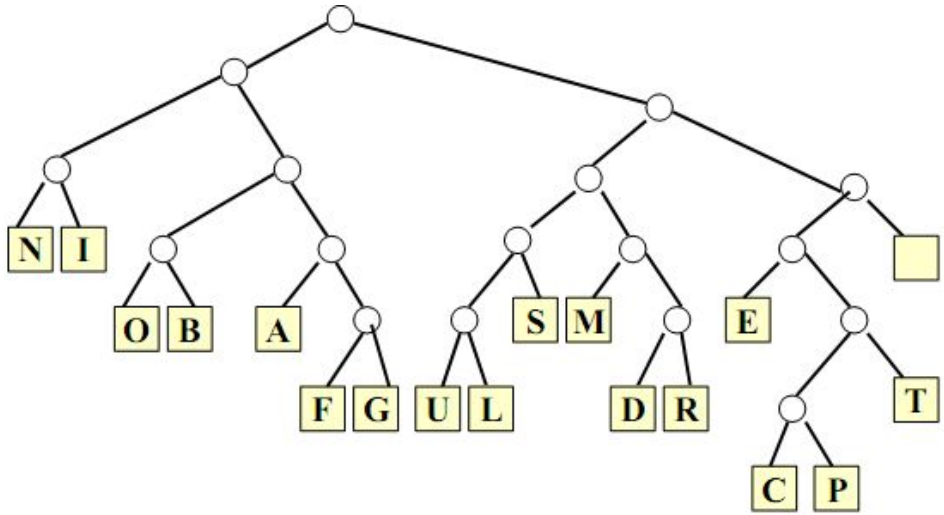
- lösche das Wurzelelement
- füge das letzte Blatt an der Wurzel ein
- stelle die Heap-Eigenschaft wieder her *(heapify)*

Aufwand:

- Min-Lesen O(1)
- Min-Löschen O(log n)
- Einfügen O(log n)

Trie für die Huffman-Codierung von “A SIMPLE STRING ...”

	_	A	B	C	D	E	F	G	I	L	M	N	O	P	R	S	T	U
k	0	1	2	3	4	5	6	7	9	12	13	14	15	16	18	19	20	21
count[k]	11	3	3	1	2	5	1	2	6	2	4	5	3	1	2	4	3	2

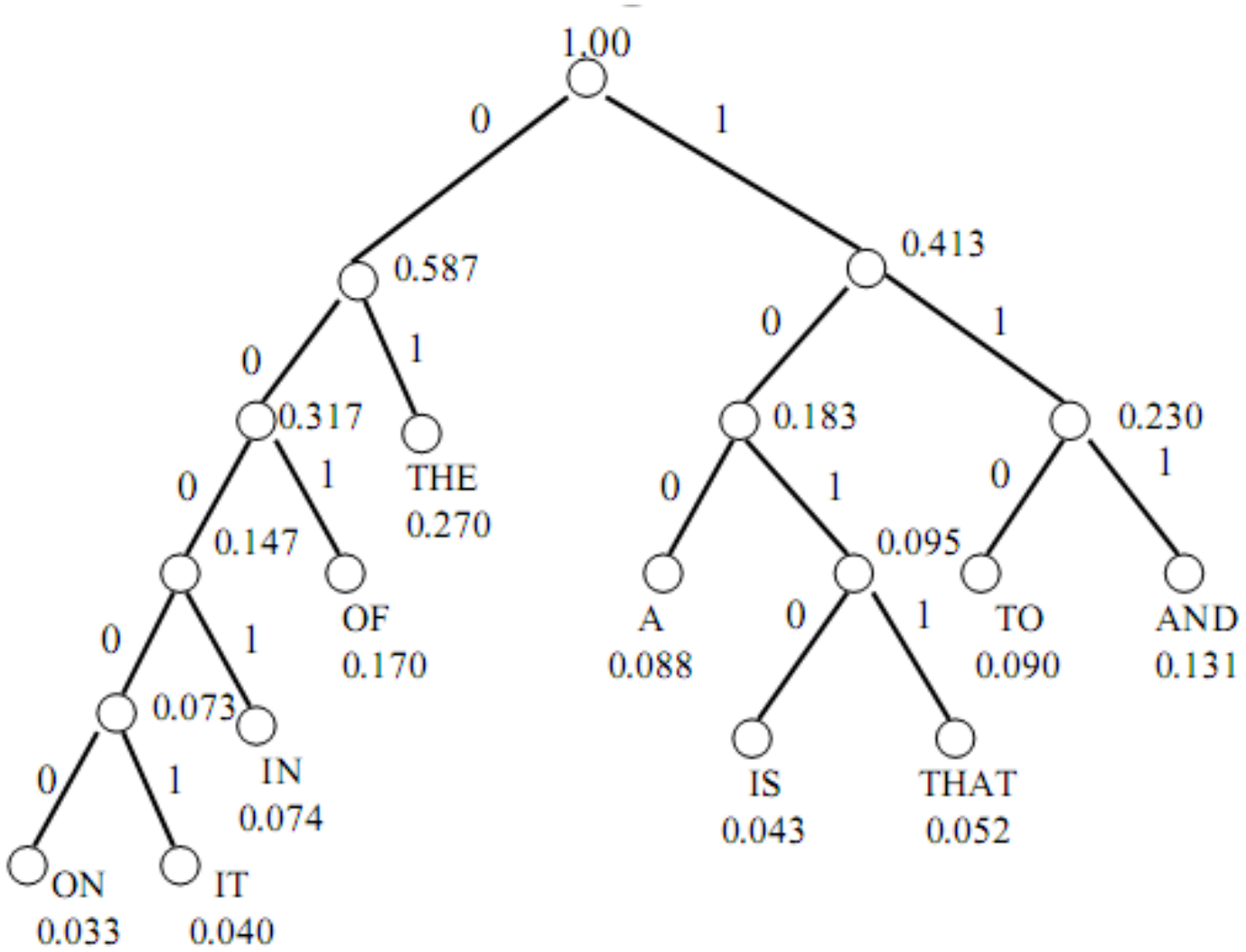


Ableitung des eigentlichen Codes aus dem Trie: Kantenlabel 0 nach links und 1 nach rechts, lese Code eines Zeichens auf Pfad von Wurzel zu Zeichen ab.

Huffman-Codierung für Wörter der Englischen Sprache

Codierungs-Einheit	Wahrscheinlichkeit des Auftretens	Code-Wert	Code-Länge
the	0.270	01	2
of	0.170	001	3
and	0.137	111	3
to	0.099	110	3
a	0.088	100	3
in	0.074	0001	4
that	0.052	1011	4
is	0.043	1010	4
it	0.040	00001	5
on	0.033	00000	5

Huffman-Codierungs-Baum für englische Wörter



Häufigkeitsverteilung deutscher Bigramme

en	4.47	*nw*	0.55	*ab*	0.25	*nm*	0.16
er	3.40	*us*	0.54	*il*	0.25	*pe*	0.16
ch	2.80	*nn*	0.53	*mm*	0.25	*rl*	0.16
nd	2.58	*nt*	0.52	*nz*	0.25	*sm*	0.16
ei	2.26	*ta*	0.51	*sg*	0.25	*sp*	0.16
de	2.14	*eg*	0.50	*sw*	0.25	*th*	0.16
in	2.04	*eh*	0.50	*rn*	0.24	*wo*	0.16
es	1.81	*zu*	0.50	*ro*	0.24	*af*	0.15
te	1.78	*al*	0.49	*ea*	0.23	*lu*	0.15
ie	1.76	*ed*	0.48	*fr*	0.23	*mu*	0.15
un	1.73	*ru*	0.48	*sd*	0.23	*no*	0.15
ge	1.68	*rs*	0.47	*tt*	0.23	*nv*	0.15
st	1.24	*ig*	0.45	*tw*	0.23	*rf*	0.15
ic	1.19	*ts*	0.45	*gr*	0.22	*ut*	0.15
he	1.17	*ma*	0.43	*tz*	0.22	*br*	0.14
ne	1.17	*sa*	0.43	*fe*	0.21	*ez*	0.14
se	1.17	*wa*	0.43	*gt*	0.21	*ho*	0.14
ng	1.07	*ac*	0.42	*rh*	0.21	*ka*	0.14
re	1.07	*eu*	0.42	*ds*	0.20	*os*	0.14
au	1.04	*so*	0.41	*du*	0.20	*bl*	0.13
di	1.02	*ar*	0.40	*mi*	0.20	*dw*	0.13
be	0.96	*tu*	0.40	*nb*	0.20	*ep*	0.13
ss	0.94	*ck*	0.37	*nk*	0.20	*hm*	0.13

Häufigkeitsverteilung deutscher Trigramme

ein	1.22	*ese*	0.27	*hre*	0.18	*nne*	0.14	*auc*	0.11
ich	1.11	*auf*	0.26	*hei*	0.18	*nes*	0.14	*als*	0.11
nde	0.89	*ben*	0.26	*lei*	0.18	*ond*	0.14	*alt*	0.11
die	0.87	*ber*	0.26	*nei*	0.18	*oen*	0.14	*eic*	0.11
und	0.87	*eit*	0.26	*nau*	0.18	*sdi*	0.14	*esc*	0.11
der	0.86	*ent*	0.26	*sge*	0.18	*sun*	0.14	*enh*	0.11
che	0.75	*est*	0.26	*tte*	0.18	*von*	0.14	*eil*	0.11
end	0.75	*sei*	0.26	*wei*	0.18	*bei*	0.13	*fen*	0.11
gen	0.71	*and*	0.25	*abe*	0.17	*chl*	0.13	*gan*	0.11
sch	0.66	*ess*	0.25	*chd*	0.17	*chn*	0.13	*hte*	0.11
cht	0.61	*ann*	0.24	*des*	0.17	*chw*	0.13	*iea*	0.11
den	0.57	*esi*	0.24	*nte*	0.17	*ech*	0.13	*ieb*	0.11
ine	0.53	*ges*	0.24	*rge*	0.17	*edi*	0.13	*nli*	0.11
nge	0.52	*nsc*	0.24	*tes*	0.17	*enk*	0.13	*rda*	0.11
nun	0.48	*nwi*	0.24	*uns*	0.17	*eun*	0.13	*rsc*	0.11
ung	0.48	*tei*	0.24	*vor*	0.17	*enz*	0.13	*std*	0.11
das	0.47	*eni*	0.23	*dem*	0.16	*hau*	0.13	*sst*	0.11
hen	0.47	*ige*	0.23	*hin*	0.16	*ite*	0.13	*tre*	0.11
ind	0.46	*aen*	0.22	*her*	0.16	*ief*	0.13	*uss*	0.11
enw	0.45	*era*	0.22	*lle*	0.16	*imm*	0.13	*all*	0.10
ens	0.44	*ern*	0.22	*nan*	0.16	*ihr*	0.13	*aft*	0.10
ies	0.44	*rde*	0.22	*tda*	0.16	*iss*	0.13	*bes*	0.10
ste	0.44	*ren*	0.22	*tel*	0.16	*kei*	0.13	*dei*	0.10
ten	0.44	*tun*	0.22	*ueb*	0.16	*mei*	0.13	*erf*	0.10
ere	0.43	*ing*	0.21	*ang*	0.15	*nsi*	0.13	*ess*	0.10
lic	0.42	*sta*	0.21	*cha*	0.15	*nem*	0.13	*esw*	0.10

Künstliche Sprache (nach Kumpfmüller)

Einergruppen (Buchstabenhäufigkeit)

EME GKNEET ERS TITBL BTZENFNDGBGD EAI E LASZ
BETEATR IASMIRCH EGEOM

Zweiergruppen (Paarhäufigkeit)

AUSZ KEINU WONDINGLIN DUFRN ISAR STEISBERER ITEHM
ANORER

Dreiergruppen

PLANZEUNDGES PHIN INE UNDEN ÜBBEICHT GES AUF ES SO
UNG GAN DICH WANDERSO

Vierergruppen

ICH FOLGEMÄSZIG BIS STEHEN DISPONIN SEELE NAMEN

Fragen zur Kompression

- Welche Arten von Redundanz können Lauflängencodierung und Huffmancodierung ausnutzen bzw. nicht ausnutzen?
- Welches Verfahren sollte wann benutzt werden?
- Welches Verfahren ist gut für Text?
- Sind Kombinationen, d.h. die Hintereinanderausführung von Kompressions-Verfahren sinnvoll?

- Verschiedene Kompressionsverfahren arbeiten unterschiedlich gut bei Daten unterschiedlichen Typs (Bilder, numerische Daten, Text).
- **Schlussfolgerung: Wir sollten uns für weitere Verfahren interessieren!**

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

Historischer Exkurs (LZ-Familie)

1977	Abraham Lempel und Jacob Ziv erfinden den Kompressionsalgorithmus LZ77
1983	Terry A. Welch (Sperry Corporation - später Unisys) patentiert LZW (Variante von LZ78)
1987	CompuServe veröffentlicht GIF als freie und offene Spezifikation (Version 87a)
1989	Vorstellung von GIF 89a
1993	Unisys informiert CompuServe über die Verwendung ihres patentierten LZW-Algorithmus in GIF
1994	Unisys gibt öffentlich bekannt, Gebühren für die Verwendung des LZW-Algorithmus einzufordern
1995	die PNG Gruppe wird gegründet, erste PNG Bilder werden ins Netz gestellt, die PNG-Spezifikation 0.92 steht im W3C
1997	PNG-Unterstützung in Netscape 4.04 und IE 4.0

Kompression nach Burrows-Wheeler-Transformation

- Michael Burrows und David Wheeler, Mai 1994
- **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)
- Die Kompression ist sehr gut geeignet für Text, da Kontext berücksichtigt wird.
- Verwendet in `bzip`

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**	

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

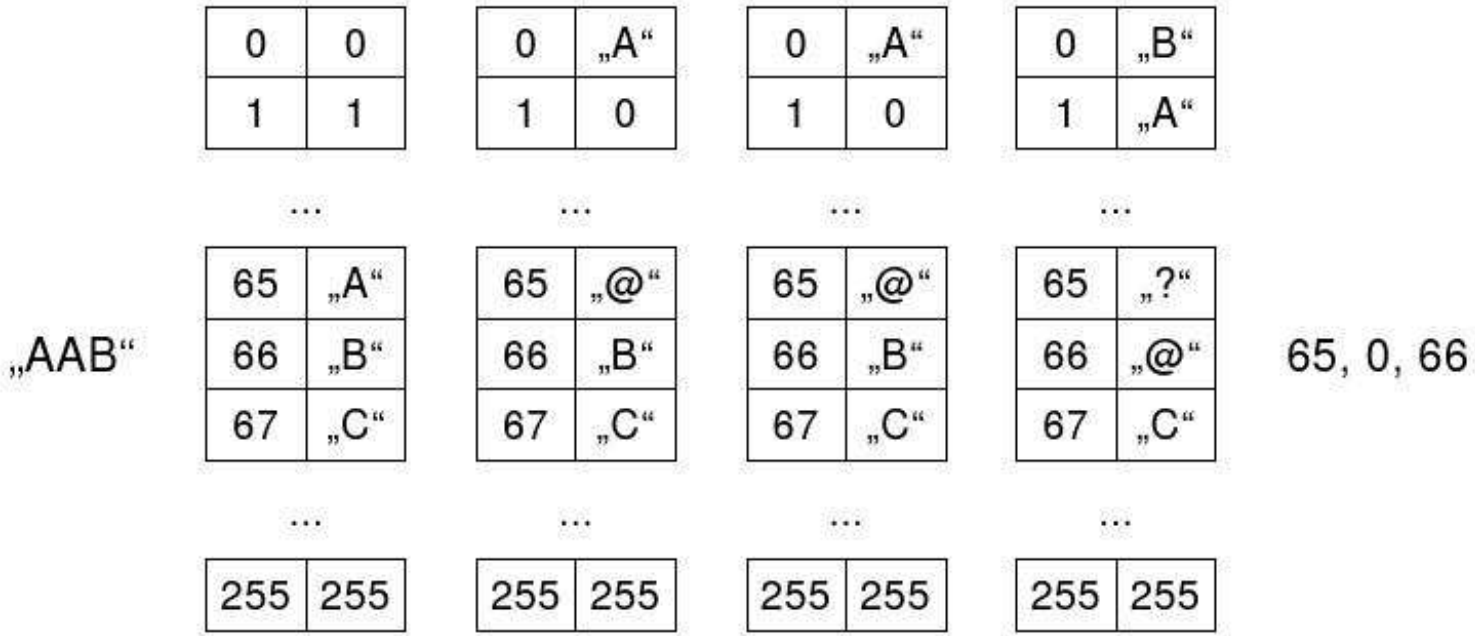
Idee: Codiere Runs von Buchstaben mit möglichst kleinen Zahlen

- 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

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', erlaubt gute Lauflängen-Kompression
- Huffman-Kodierer ermöglicht kurze Bitfolgen für häufige Zeichen

bzip-Funktionen:

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

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

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

Definition (Shannon)

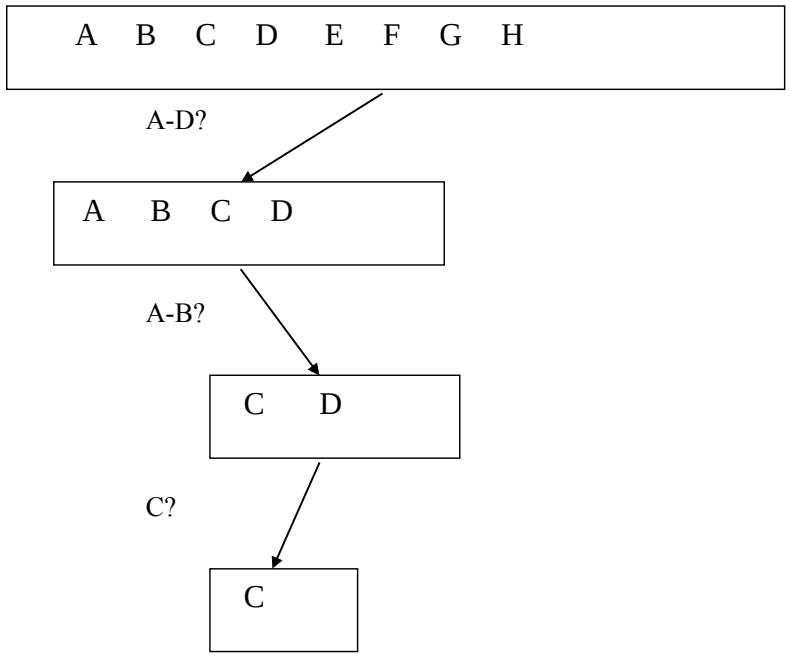
Entscheidungsinformation: Anzahl optimal gewählter binärer Entscheidungen zur Ermittlung eines Zeichens in einem Zeichenvorrat
Entropie: der mittlere Informationsgehalt eines Textes
Redundanz: Anteil einer Nachricht, der keine Information enthält

- 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

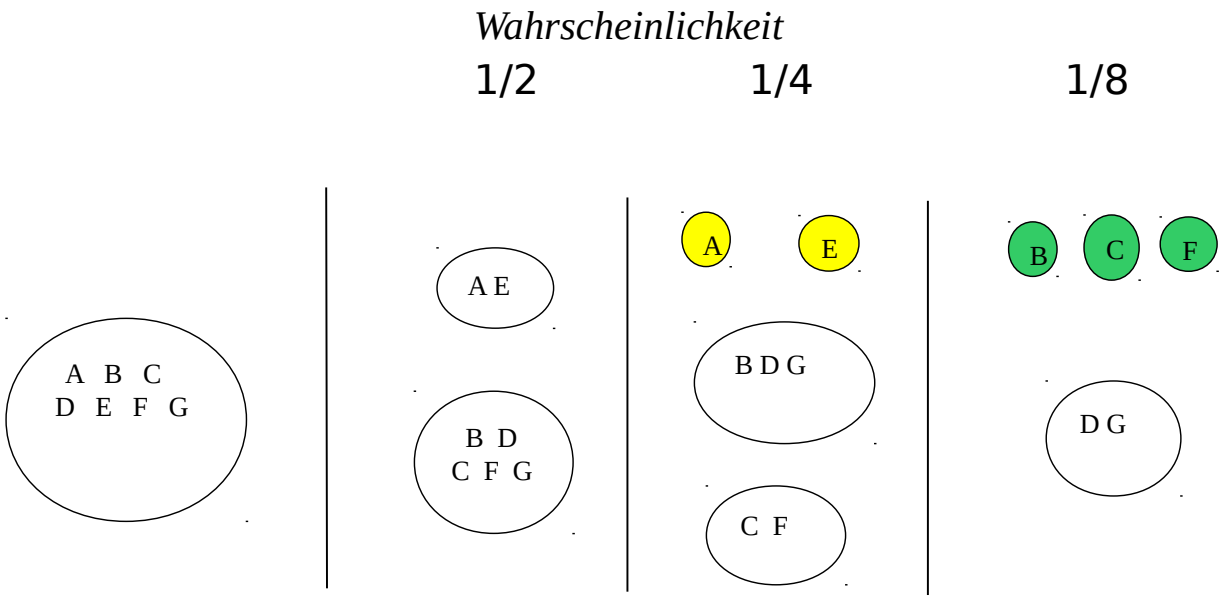
Gegeben seien 8 Zeichen. Nach maximal wieviel Schritten ist ein Zeichen gefunden?

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

BEIREDUZIERTERREDUNDANZWIRDDASLESENSEHRVIELMüHSA-MER

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