Datenkompression

- Übersicht
- Literatur
- Klassifikation:

ausgewählte Verfahren:

- Lauflängenkodierung
- Huffman Kodierung, arithmetische Kodierung
- Lempel-Ziv Kodierung und Varianten
- Burrows-Wheeler Transformation

- Wavelets

- weitere Verfahren später (Audio, Image, Video)

Ad-Hoc Kodierung: Morse, Braille, ...

A	.-	N	-.	1	.----
B	-...	O	---	2	..---
C	-.-.	P	.--.	3	...--
Ch	----	Q	--.-	4	....-
D	-..	R	.-.	5	.....
E	.	S	...	6	-....
F	..-.	T	-	7	--...
G	--.	U	..-	8	---..
H	....	V	...-	9	----.
I	..	W	.--	0	-----
J	.---	X	-..-		
K	-.-	Y	-.--		
L	.-..	Z	--..		
M	--				

Period	.-.-.-
Comma	--..--
Colon	---...
Question mark	..--..
Apostrophe	.----.
Hyphen	-....-
Dash	-..-.
Parentheses	-.--.-
Quotation marks	.-..-.

If the duration of a dot is taken to be one unit, then that of a dash is three units. The space between the dots and dashes of one character is one unit, between characters is three units, and between words six units (five for automatic transmission). To indicate that a mistake has been made and for the receiver to delete the last word, send "........" (eight dots).

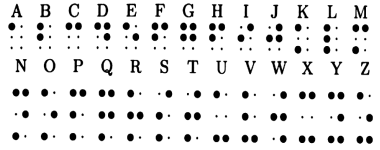


Table 1.1: The 26 Braille Letters.

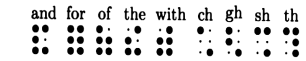
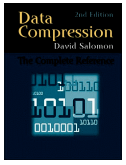


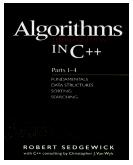
Table 1.2: Some Words and Strings in Braille.

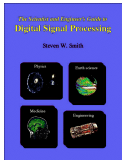
- zwei wichtige Beispiele für "frühe" Codes
- Morse: häufige Symbolei ("e") mit kurzen Codes
- Braille: alle 64 Codes verwendet, keine Fehlererkennung

(Salomon)

Datenkompression: Literatur







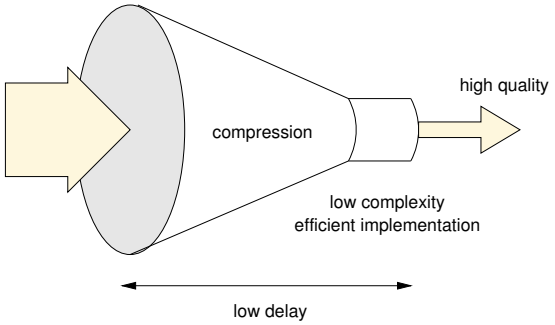
David Salomon, Data Compression - The Complete Reference, Springer 2000
Robert Sedgewick, Algorithms in C++, Addison-Wesley, 1998
Stephen W. Smith, Guide to Digital Signal Processing, California Technical Publ., 1997
online als PDF verfügbar: www.dspguide.com

Mark Nelson, Data Compression with the Burrows-Wheeler Transform,
Dr. Dobbs Journal, 9/1996
F. Bauernöppel, Verfahren und Techniken zur Datenkompression, c't 10/1991
M. Tamm, Packen wie noch nie - BWT-Algorithmus, c't 16/2000, 194

Multimedia: Datenmengen

Medium:	Datenmenge:	Komprimierung:	
Text	80 x 60 Zeichen 4.8 KB	Huffman 1:2 PKZIP 1:3	
Bild	640x480x3 = 900 KByte	Fax 1:10 JPEG 1:15	
Grafik	500 Linien		
Sprache	1 Kanal, 8Khz 64 Kbps, 480 KB/min	GSM 1:8	
MIDI	15 KB/min		
Musik	stereo, 44.1 KHz 1.44 Mbps, 10 MB/min	MP3 1:10	
Video	640x480x3x25 1.3 GB/min	MPEG2 1:60	

Datenkompression: Kriterien



- gute Kompressionsrate, gute Qualität
- geringe Latenz (Verzögerung)
- geringer Bedarf an Rechenleistung und Speicherbedarf

Klassifikation:

"direct coding"		PCM
"entropy coding"	repetitive sequence suppresion	zero suppresion
		run-length encoding
	statistical encoding	pattern substitution
		Huffman encoding
"source coding" (e.g. speech)	transform encoding	FFT
		DCT
		. . .
	differential encoding	DPCM
		delta modulation
		ADPCM
	vector quantization	general / fractal / . . ..
"channel coding"		HDLC, ...

Entropie- vs. Quellenkodierung

Entropiekodierung:

- Eigenschaften der Datenquelle werden ignoriert
- Signalwiederholungen entfernen
- statistische Verfahren, z.B. Huffman-Kodierung
- verlustfrei und reversibel
- geringe Kompressionsfaktoren (z.B. ca. 2 für Audiodaten)

Quellenkodierung (source encoding):

- Eigenheiten der Datenquelle / senke berücksichtigen
- z.B. Frequenzgang / Maskierung / Rauschschwellen des Ohrs
- verlustfrei
- verlustbehaftet für bessere Kompression (z.B. MP3 bis ca 10:1)

Codec

- verlustlose Datenkompression:

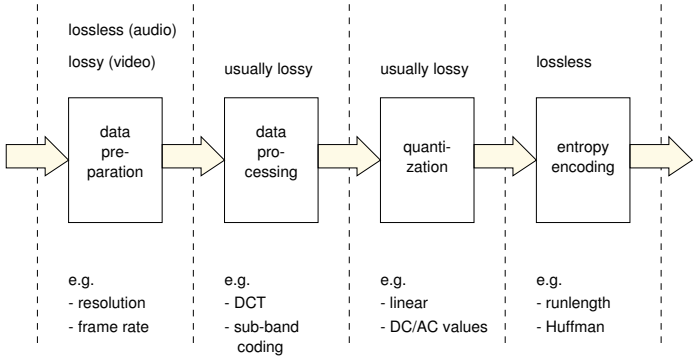
  decode(encode(x)) = x

- jedes Verfahren benötigt Paar aus Coder und Decoder
  := "CODEC"

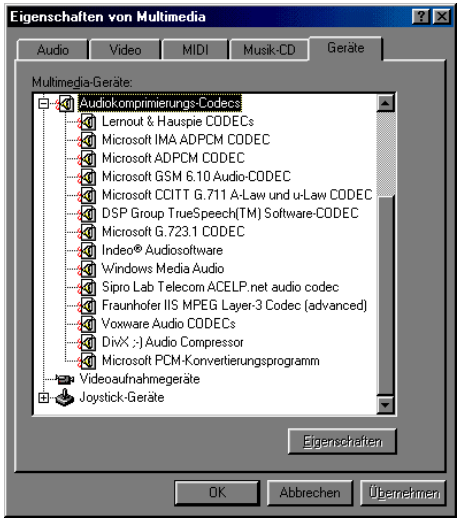
vollständige Abstraktion und Kapselung möglich:

- decode1(decode2(kanal(encode2(encode1(x))))) = x
- äußere / innere Schichten brauchen nicht vom Codec zu wissen
- beliebig tiefe Schachtelung
- siehe Windows / Java Media Framework

CODECs: Verkettung



Codecs: Beispiel Windows 9x



Windows-Systemsteuerung:

- sammelt CODECS
- Audio, Video, MIDI

- je nach installierter SW
- hier: 14 Audio-Codecs
- Sprache vs. Musik

Datenkompression: mehrfach?

- komprimierte Daten beliebig weiter komprimierbar?
- diverse Patentanmeldungen dazu :-)

- aber nicht möglich (sonst jede Datei auf 1 Bit komprimierbar...)
- nur mit "Tricks" (z.B. Daten in Dateinamen speichern)

=> jeder Algorithmus expandiert bestimmte Dateien

Achtung: Verschlüsselung beseitigt Redundanzen der Daten:

- Verschlüsselung nur als letzter Schritt
- nach allen Kodierungsschritten
- verschlüsselte Daten nicht komprimierbar

(siehe Salomon oder Sedgewick)

Transformationen

- Sinuston, Darstellung im Zeitbereich und Frequenzbereich
- Darstellung im Frequenzbereich offenbar weit effizienter

wichtige Beispiele für Transformationen:

- Burrows-Wheeler (für Textdateien)
- Fouriertransformation (Audio, Bilder)
- Cosinustransformation
- Wavelet-Transformation

Datenkompression: Szenarien

Backup, Archivierung:

- möglichst gute Kompression, schneller Encoder
- Komplexität und Geschwindigkeit des Decoders egal

Software-Archive (JAR, Zip):

- möglichst schneller Decoder, gute Kompression
- Encoder wird selten benutzt,

Datenübertragung (Modem):

- Echtzeitanforderungen, möglichst gute Performance
- Notwendigkeit zum Umgang mit bereits komprimierten Daten

CD/DVD, weak-signal Kommunikation:

- möglichst gute Fehlererkennung und -korrektur

Informationsgehalt, Redundanz

The entropy of the data depends on the individual probabilities P_i, and is smallest when all n probabilities are equal. This fact is used to define the redundancy R in the data. It is defined as the difference between the entropy and the smallest entropy. Thus

$$R = \left(-\sum_1^n P_i \log_2 P_i \right) - \log_2(1/P) = -\sum_1^n P_i \log_2 P_i + \log_2 n.$$

The test for fully compressed data (no redundancy) is thus $\sum_1^n P_i \log_2 P_i = \log_2 n$.

- kann bei bekannter Nachricht leicht berechnet werden
- oder aus bisherigen Symbolen schätzen

- gute Kodierung erreicht R ~= 0

(Shannon)

Run-Length Kodierung

2. all is too well
2. a2l is t2o we2l
2. a@2l is t@2l we@l

"Lauflängenkodierung"

- Ersetzen von wiederholten Symbolen durch Marke+Anzahl+Symbol
- lohnt nur ab Lauflänge 3
- diverse Varianten zur Kodierung des Zählers

- sehr einfach, sehr schnell
- nur für bestimmte Daten geeignet
- z.B. für S/W-Bilder
- aber nicht für normale Text, Audio, Grauwert- oder Farbbilder

Lauflängenkodierung: FAX (group 3)

Run length	White code-word	Black code-word
0	00110101	0000110111
1	000111	010
2	0111	11
3	1000	10
4	1011	011
5	1100	0011
6	1110	0010
7	1111	00011
8	10011	000101
9	10100	000100
10	00111	0000100
11	01000	0000101
12	001000	0000111
13	000011	00000100
14	110100	00000111
15	110101	000011000
16	101010	0000010111
17	101011	0000011000
18	0100111	0000001000
19	0001100	00001100111
20	0001000	00001101000
21	0010111	00001101100
22	0000011	00000110111
23	0000100	00000101000
24	0101000	00000101111
25	0101011	00000011000
26	0010011	000011001010
27	0100100	000011001011
28	0011000	000011001100
29	00000010	000011001101
30	00000011	000001101000
31	00011010	000001101001

(a)

Run length	White code-word	Black code-word
64	11011	0000001111
128	10010	000011001000
192	010111	000011001001
256	0110111	000001011011
320	00110110	000000110011
384	00110111	000000110100
448	01100100	000000110101
512	01100101	000000110000
576	01101000	000000110001
640	01100111	000000100010
704	011001100	000000100011
768	011001101	000000100100
832	011010010	000000100101
896	011010011	000000110010
960	011010100	000000110011
1024	011010101	000000110100
1088	011010110	000000110101
1152	011010111	000000110110
1216	011011000	000000110111
1280	011011001	000000101010
1344	011011010	000000101011
1408	011011011	000000101100
1472	010011000	000000101101
1536	010011001	000000101010
1600	010011010	000000101011
1664	011000	000000110010
1728	010011011	000000110101
1792	00000001000	same as
1856	00000001100	white
1920	00000001101	from this
1984	000000010010	point
2048	000000010011	
2112	000000010100	
2176	000000010101	
2240	000000010110	
2304	000000010111	
2368	000000011100	
2432	000000011101	
2496	000000011110	
2560	000000011111	

(b)

Table 2.30: Group 3 and 4 Fax Codes: (a) Termination Codes. (b) Make-Up Codes.

Auflösung:

- horizontal: 8.5 Pixel/mm
- vertikal: 3.85 / 7.7 / 15.4 Zeilen/mm
- (etwa 2200x1700 Pixel pro Seite A4)

Analyse typischer Briefe / Dokumente:

- häufig 2, 3, 4 schwarze Pixel
- oder 2 .. 7 weisse Pixel
- oder ganz weisse Zeilen ...

- ungeeignet für Bilder / Graustufen
- dafür besserer Code ("group 4")

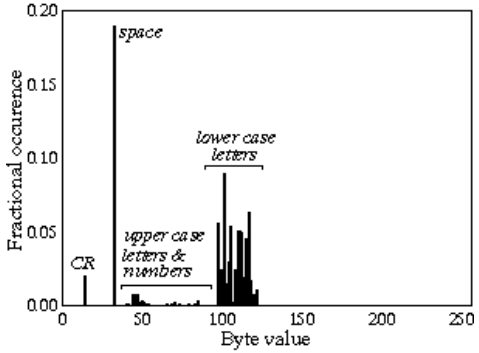
Buchstabenhäufigkeiten . . .

```
53++!305))6*;4826)4+.)4+);806*;48!8'60))85;]8*:+*8!83(88)5*!;
46(;88*96*?;8)*+(;485);5*!2:*+(;4956*2(5*)8'8*;4069285);)6
!8)4++;1(+9;48081;8:8+1;48!85;4)485!528806*81(+9;48;(88;4(+?3
4;48)4+;161;:188;+?;
```

"Here, then, we have, in the very beginning, the groundwork for something more than a mere guess. The general use which may be made of the table is obvious - but, in this particular cipher, we shall only very partially require its aid. As our predominant character is 8, we will commence by assuming it as the "e" of the natural alphabet. To verify the supposition, let us observe if the 8 be seen often in couples - for "e" is doubled with great frequency in English - in such works, for example, as "meet", "fleet", "speed", "seen", "been", "agree", etc. In the present instance we see it double no less than five times, although the cyrptograph is brief.

- Edgar Allen Poe, The Gold Bug

Beispiel: Buchstabenhäufigkeiten

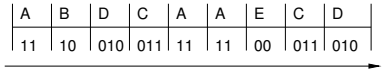


- Einzelhäufigkeiten in English: "ENATOIN SHRDLU ..."
- entsprechende Modelle auch für Paare ("qu"), Tripets, ...
- Grundidee: häufige Symbole mit kurzen Bitstrings kodieren

(Scan: Buchstabenhäufigkeiten im DSP Guide)

Huffman-Kodierung

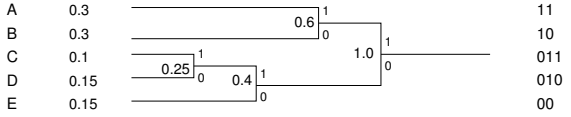
Symbole	Häufigkeit	Codewort
A	0.3	11
B	0.3	10
C	0.1	011
D	0.15	010
E	0.15	00



Ausnutzen der statistischen Eigenschaften der Daten:

- häufige Zeichen mit kurzen Bitfolgen kodieren
- seltene Symbole mit längeren (evtl. sehr langen)

Huffman: Konstruktion



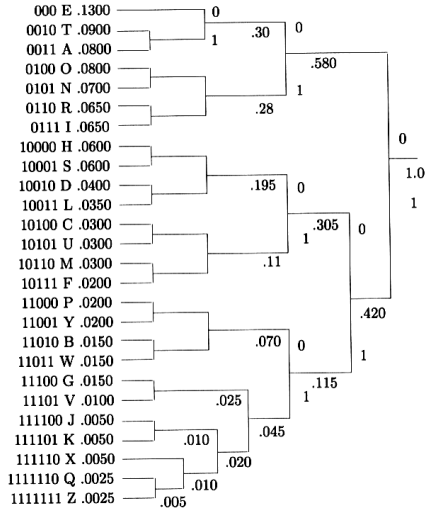
1. sortiere die Symbole nach ihrer Häufigkeit
2. kombiniere die zwei Symbole mit geringster Häufigkeit
   (bei gleichen Wahrscheinlichkeiten: zufällige Auswahl)
3. wiederhole die Konstruktion, bis alle Symbole kombiniert sind

- "bottom-up" Verfahren
- erfordert a-priori Kenntnis der Häufigkeiten

- "semi-adaptive": 2-pass Verfahren, sehr langsam
- "adaptive": Anpassen des Baums an geänderte Häufigkeiten

(siehe Salomon 2.9)

Huffman: Beispiel



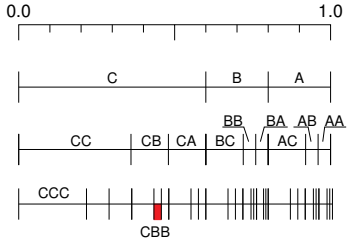
(Salomon, 2.12)

Arithmetic Coding

- Huffman-Kodierung ist nur in Spezialfällen optimal
- weil jedem einzelnen Symbol ein Code zugeordnet wird

arithmetische Kodierung:

- Zuordnung von Zeichen zu arith. Intervallen
- entsprechend der Zeichenhäufigkeiten
- rekursiv für Zeichenfolgen
- Verfahren patentiert und lizenzpflichtig



p(C) = 0.6 p(B) = 0.2 p(A) = 0.2

Beispiel-Intervall für Folge "cbb":

0.6*0.6 + 0.8*(0.48 - 0.36) = 0.456

0.6*0.6 + 0.6*(0.48 - 0.36) = 0.432

0.432 = 0.0110 0.456 = 0.0111

Binärcode für "cbb" = 0111

Shannon-Fano Kodierung

	Prob.	Steps				Final
1.	0.25	1	1			:11
2.	0.20	1	0			:10
3.	0.15	0		1	1	:011
4.	0.15	0		1	0	:010
5.	0.10	0		0	1	:001
6.	0.10	0		0	0	1 :0001
7.	0.05	0		0	0	0 :0000

Table 2.9: Shannon-Fano Example.

	Prob.	Steps			Final
1.	0.25	1	1		:11
2.	0.25	1	0		:10
3.	0.125	0	1	1	:011
4.	0.125	0	1	0	:010
5.	0.125	0	0	1	:001
6.	0.125	0	0	0	:000

2.10: Shannon-Fano Balanced Example.

- Symbole in möglichst gleichgroße Mengen einteilen
- linke (im Bild: obere) Menge bekommt Bit 1, andere Menge Bit 0
- für alle Teilmengen mit mehr als 2 Elementen wiederholen

- Verfahren ähnlich zur Huffman-Konstruktion
- aber nur optimal, wenn in jedem Schritt gleichgroße Mengen

Dictionary-Methoden

- warum Beschränkung auf Einzelsymbole?
- Verzeichnis mit häufigen Worten (Symbolfolgen) aufbauen
- Index in dieses Verzeichnis kodieren
- unbekannte Symbolfolgen in das Verzeichnis aufnehmen

- Lempel-Ziv Verfahren, Dutzende Varianten
- universell anwendbar: Texte, Programme, Bilder, ...
- sehr gute Kompressionsraten (für verlustfreie Algorithmen)

LZ77

sir sid eastman easily teases sea sick seals		
	sir sid eastman ...	(0,0,"s")
s	ir sid eastman	(0,0,"i")
si	r sid eastman	(0,0,"r")
sir	sid eastman eas...	(0,0," ")
sir	sid eastman	(4,2,"d")
sir sid eastman easily t	eases sea sick seals	(16,3,"e")

"sliding window" Methode:

- "search buffer" (links) als Dictionary, einige KByte
- Tokens mit Index ins Dictionary und nächstes Symbol schreiben

LZW-Algorithmus

```
for (i=0; i < 255; i++) {
  append i as a 1-symbol to the dictionary
}
append null to the dictionary;
di = dictionary index of null;
repeat
  read(ch);
  if <<<di,ch>> is in the dictionary then
    di = dictionary index of <<<di,ch>>;
  else
    output(di);
    append <<di,ch>> to the dictionary;
    di = dictionary index of ch;
  endif;
until end-of-input;
```


LZW-Verfahren

- Welsh, 1984
- Dictionary mit Einzelsymbolen (z.B. 256 Bytes) initialisieren
- unbekannte Strings in Dictionary aufnehmen
- dabei Strings maximaler Länge versuchen

- was passiert bei vollem Dictionary?
- diverse Varianten: Dictionary löschen, LRU, ...
- Einsatz u.a. für ZIP, gzip, GIF, TIFF; ...

LZW: Beispiel

0	NULL	256	si
1	SOH	257	ir
	...	258	r_
32	space	259	_s
	...	260	sid
97	a	261	d_
98	b	262	_e
99	c	263	ea
		...	
255	255	4095	

- Strings wachsen nur langsam (max. 1 Zeichen)

BWT: Burrows-Wheeler Transformation

- Eingangsdaten in (großen) Blöcken bearbeiten
- z.B. bzip2: 100 .. 900 KByte Blockgröße

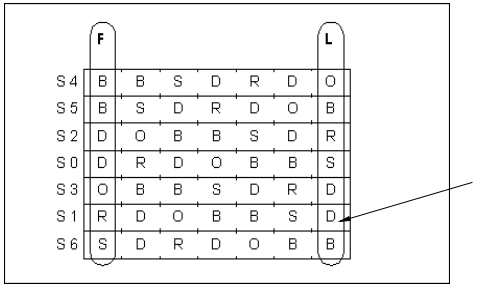
- Daten so umordnen daß ähnliche Strings entstehen
- anschließend normale statistische Kodierung
- völlig neuartiges Prinzip, ursprüngliche Idee (Wheeler 1983)

- adaptiert automatisch an statistische Eigenschaften der Daten
- Algorithmus funktioniert für Texte, Programme, beliebige Daten

gute Beschreibungen:
M. Nelson, Data compression with the BWT, Dr. Dobbs. 9/1996
M. Tamm, Packen wie noch nie, c't 16/2000

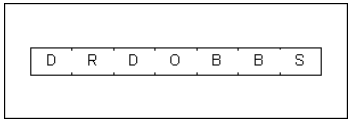
(Burrows, Wheeler: A block sorting lossless data compression algorith, DEC report 124, 1994)

BWT: Puffer, Rotation

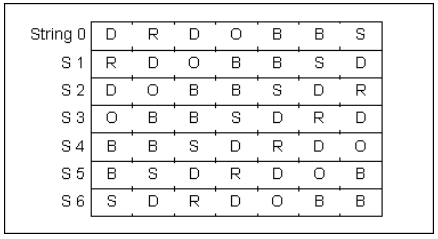


- Sortieren der Matrix, Aufbau eines Indexvektors
- geeignete Vergleichsfunktion wie strcmp() mit wrap-around
- markierte Spalten F (first) und L (last) der Originaldaten
- Zeichen in L sind jeweils die Vorgänger der Zeichen in F
- Ausgabe der BWT: Spalte L und "Primärindex" (hier 5)

BWT: Aufbau der Matrix



- Eingabedaten zu einer Matrix anordnen
- jede Zeile der Matrix um ein Zeichen rotiert



(aber 100000 x 100000 Matrix ist unhandlich)

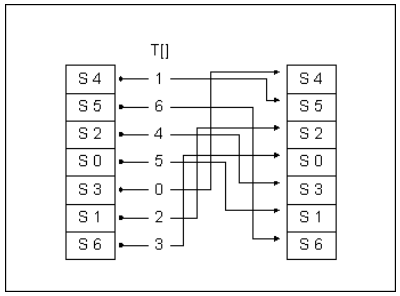
BWT: Puffer, sortiert

```
L: F.........................

t: hat redistributors of a free
t: hat refer to this License an
t: hat system in reliance on co
w: hat the Program does.\n\n  1
t: hat there is no warranty (or
t: hat there is no warranty for
w: hat they have is not the ori
t: hat they, too, receive or ca
t: hat users may redistribute t
t: hat version or of any later
t: hat what they have is not th
t: hat work are not derived fro
```

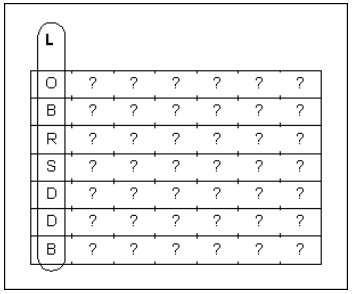
- (BWT eines Ausschnitts aus der GPL)

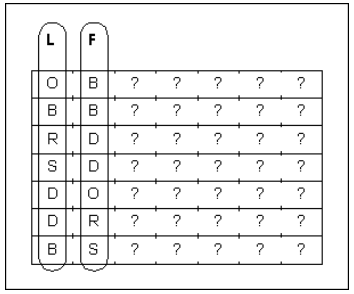
BWT: Transformationsvektor



- T[i] enthält die Zeile, die S[i+1] enthält
- Zeile 3 enthält S0, Zeile 5 enthält S1, also T[3] = 5
- Zeile 2 enthält S2, also T[5] = 2, usw.
- im Beispiel also T = { 1, 6, 4, 5, 0, 2, 3 }
- aber: Berechnung von T aus L ?

BWT: Rekonstruktion von T





- Wiederherstellung von F:
- L ist gegeben, F war sortiert: einfach die Zeichen in L sortieren
- Zeichen "O": offenbar gilt T[4] = 0
- F sortiert: T[1] = 0, T[6] = 1, usw.

BWT: Move-to-Front Kodierung

- verwaltet Liste mit 256 Werten
- Ausgabe eines Zeichens:
  Index des Zeichens in der Liste ausgeben
  und Zeichen nach vorne (Index 0) verschieben
- Beispiel: "tttWtwttt" (aus sortierter GPL)
  Ausgabe: { 116, 0, 0, 88, 1, 119, 1, 0, 0 }
- anschließend noch Huffman- oder arithmetische Kodierung

BWT: Gesamtablauf

```
Compressing a file using the demo programs:

RLE input-file | BWT | MTF | RLE | ARI > output-file

A brief description of each of the programs follows:

RLE.CPP
            This program implements a simple run-length encoder. If the input
            file has many long runs of identical characters, the sorting
            procedure in the BWT can be degraded dramatically.
BWT.CPP
            The standard Burrows-Wheeler transform is done here.  This
            program outputs repeated blocks consisting of a block size integer,
            a copy of L, the primary index, and a special last character index.
            This is repeated until BWT.EXE runs out of input data.
MTF.CPP
            The Move to Front encoder.
RLE.CPP
            The fact that the output file is top-heavy with runs
            containing zeros means that applying another RLE pass
            to the output can improve overall compression.
ARI.CPP
            This is an order-0 adaptive arithmetic encoder,
            directly derived from the code published by Witten and Cleary
            in their 1987 CACM article.
```

BWT: Vergleich mit anderen Verfahren

Performance Figures

The table below shows obtained bit/byte for some different universal coding utilities and files. I have chosen pretty large files to lessen the boundary-effects at the start.

File × Utility -> bit/byte	bzip	dmc	gzip	zip	arj	lha	lharc	compress
bible.tar (5140480 bytes)	1.80	1.83	2.53	2.53	2.60	2.80	2.97	2.89
netscape (4501956 bytes)	3.29	3.45	3.51	3.51	3.51	3.58	3.72	4.89
gcc-2.7.2.1.tar (7090289 bytes)	1.49	1.61	2.02	2.01	2.02	N/A	N/A	2.99

The file *bible.tar* consists of selected texts from the bible (In swedish). See Project Runeberg. Notice the superior performance of bzip and dmc for this kind of source! The perhaps not so well-known coder *dmc* uses dynamic markov modelling and arithmetic coding and was written by Gordon V. Cormack. It performs almost as good as bzip but is slower.

The file *netscape* is a stripped binary. bzip performs best also here but the difference is not so big, althogh note the lousy performance of the standard UNIX utility compress. (Uses LZW technique).

The file *gcc-2.7.2.1.tar* consists of lot of C-code. Also here bzip is outstanding. UNIX compress has to use double as many bits as bzip...

The following programs and flags were used to generate the different codes:

- *bzip -9* ver. 0.21
- *dmc 50000000* (That's a large buffer in case you wondered) ver 0.0.0
- *gzip -9* ver 1.2.4
- *zip -9* ver 2.0.1 for UNIX (performs better than *pkzip -ex*)
- *arj -em* ver 2.30
- *lha* LHa ver. 1.00 for UNIX
- *lha o* LHarc ver. 1.02 for UNIX
- *compress* UNIX (N)compress ver. 4.2.4

Wavelets: Beispielfunktionen

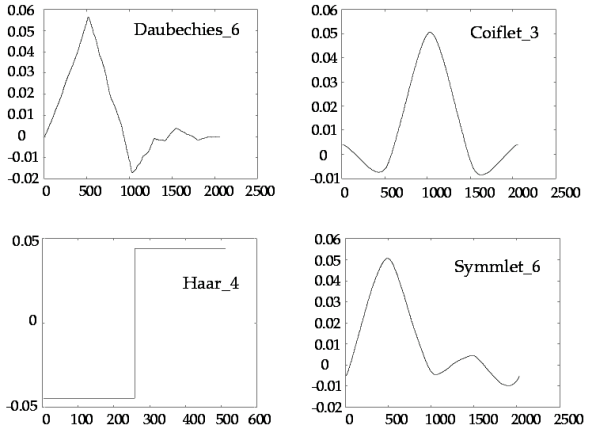


Fig. 4. Several different families of wavelets. The number next to the wavelet name represents the number of vanishing moments (A stringent mathematical definition related to the number of wavelet coefficients) for the subclass of wavelet. These figures were generated using WaveLab.

(Amara Graph, Introduction to Wavelets)

Wavelet-Transformation

- Wavelets := "kleine Wellen"
- "lokalisierte", skalierbare Basisfunktionen

$$\Phi_{(s,l)}(x) = 2^{-\frac{s}{2}}\Phi\left(2^{-s}x - l\right)$$

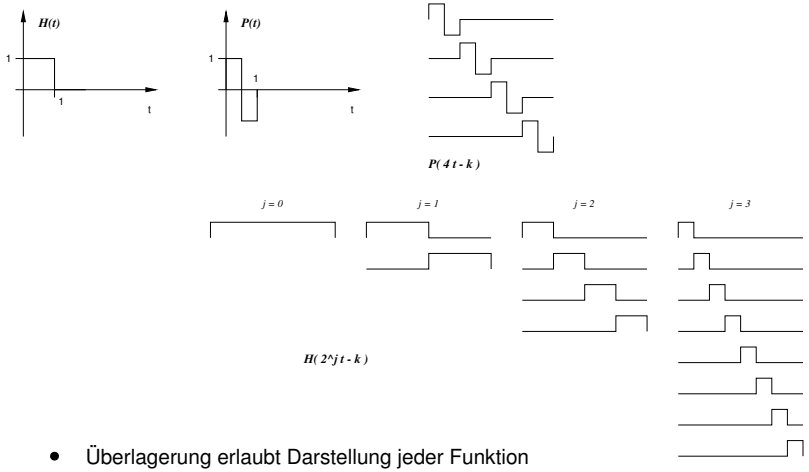
- diverse Basisfunktionen möglich
- Eigenschaften können "maßgeschneidert" werden:
- lineare Transformation:

$$W(x) = \sum_{k=-1}^{N-2} (-1)^k c_{k+1}\Phi(2x+k)$$

- Normierungsbedingung:

$$\sum_{k=0}^{N-1} c_k = 2, \quad \sum_{k=0}^{N-1} c_k c_l = 2\delta_{l,0}$$

Wavelets: Haar-Wavelet



- Überlagerung erlaubt Darstellung jeder Funktion

(s. Salomon 5.6)