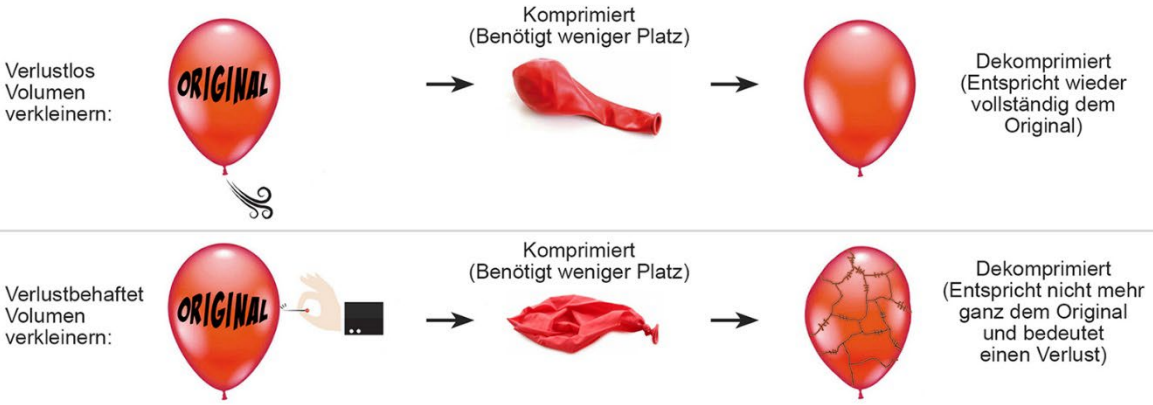
DATEN KOMPRIMIEREN THEORIE

Beim Komprimieren unterscheiden wir die **verlustlose** Komprimierung von der **verlustbehafteten** Komprimierung:



- **Verlustlose Komprimierung**: Es gibt viele verschiedene theoretische Konzepte und Algorithmen für verlustlose Komprimierung wie z.B. VLC (Variable Length Coding) mit z.B. dem Morsecode oder Huffman, RLC (Run Length Coding), lexikalische Verfahren wie z.B. LZ77/LZ78(Lempel-Ziv), LZW (Lempel-Ziv-Welch) oder Deflate, Transformationsverfahren, die selber allerdings keine Datenkompression durchführen wie z.B. BWT (Burrows-Wheeler-Transformation) und viele weitere. Allen gemeinsam ist, dass komprimierte Daten verlustlos dekomprimiert werden können. Bsp.: Text, Applikationen, Datenbanken etc

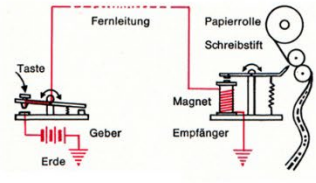
- **Verlustbehaftete Komprimierung**: Bei den wirklichen Schwergewichten wie sie im Bereich Multimedia (Bild, Film, Ton) anzutreffen sind, genügen verlustlose Komprimierungsverfahren alleine nicht. Man versucht mit geeigneten Verfahren, in einer Datei den Informationsgehalt derart zu reduzieren, dass das subjektive Empfinden des Datei-Konsumenten die Datenreduktion nicht oder kaum wahrnimmt. Unter anderem gelingt dies mit Transformationsverfahren, die zwar selber noch keine Datenreduktion herbeiführen, einem darauffolgenden Komprimierungsverfahren wie z.B. Huffman, RLC, LZ78 etc. aber entsprechendes Potential bieten. Bsp.: DCT (Discrete Cosine Transformation) das unter anderem bei JPG zur Anwendung kommt. Verlustbehaftet komprimierte Daten entsprechen nach der Dekomprimierung nicht mehr dem Original.

  (Die verlustbehaftete Komprimierung bei Multimedia wird zusammen mit der Bildcodierung in einem späteren Kapitel behandelt.)

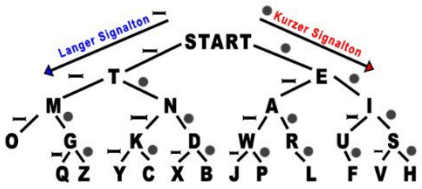
Einführung:

Eine Variante, mit der Mitteilungen effizient ausgetauscht werden können, ist uns wahrscheinlich allen schon längst bekannt: **Der Morsecode**. Beim Morsecode handelt es sich um einen «**Variable Length Code**» oder kurc **VLC** was nichts anderes heisst, als «Variable Codelänge» und meint, dass einer festen Anzahl an Quellsymbolen jeweils Codewörter mit variabler Länge zugeordnet sind. Ganz im Gegensatz zu z.B. ASCII-Code, wo einem Quellsymbol jeweils ein Codewort mit fixer Länge von 8 Bit zugeordnet wird.
Nun aber zurück zum Morsecode:

Der Morsecode, erfunden um 1838 von Samuel Morse und Alfred Lewis Vail, ist ein Verfahren, um in Seefunk und Telegrafie elektrisch, optisch oder akustisch verlustlos komprimiert Buchstaben, Zahlen und weitere Zeichen zu übermitteln. Der Morsecode kann man als Codetabelle oder als binären Baum darstellen.



A	●–	J	●–––	S	●●●		
B	–●●●	K	–●–	T	–		
C	–●–●	L	●–●●	U	●●–		
D	–●●	M	––	V	●●●–		
E	●	N	–●	W	●––		
F	●●–●	O	–––	X	–●●–		
G	––●	P	●––●	Y	–●––		
H	●●●●	Q	––●–	Z	––●●		
I	●●	R	●–●				



Um ihre Morse-Skills auf die Probe zu stellen, hier gleich ein Beispiel dazu:
https://www.juergarnold.ch/Kompression/Morsecode.wav Was wird uns da mitgeteilt?

Wo liegt «der Hund begraben»?:
Die folgende Codesequenz « `..–....–` » kann so vieles heissen, wie z.B

- IDEA: `..(I)` `–..(D)` `.(E)` `.–(A)`
- USA: `..–(U)` `...(S)` `.–(A)`
- UHT: `..–(U)` `....(H)` `–(T)` (UHT=**U**ltra**H**igh**T**emperatur)
- FV: `..–.(F)` `...–(V)` (FV=**F**ussball**v**erein)

Also keinesfalls eindeutig!

Der Nachteil beim Morsecode liegt nämlich darin, dass es ohne spezielles **Trennzeichen** oder Delimiter (Beim Morsecode eine kurze Pause) **Missverständnisse** geben kann, wo das Zeichen beginnt und wo es endet. Dieses Problem besteht z.B. bei der nachfolgenden Huffman-Kodierung nicht, weil hier die Eigenschaft erfüllt wird, dass **kein Codewort der Beginn eines anderen Codewortes** darstellt.

VLC am Beispiel Huffman (Huffmancode):

Um einen Huffman-Code für eine **feste Anzahl von Quellsymbolen** zu entwickeln, baut man Schritt für Schritt einen **binären Baum** auf, ähnlich dem beim Morsecode. Wie das geschieht, soll das folgende Beispiel zeigen, wo als Quellsymbole nur die Buchstaben inkl. Häufigkeit für das Wort **ERDBEERE** Verwendung finden:

Wir stellen fest, dass das in ERDBEERE am **häufigsten auftretende Quellsymbol E das kürzeste Codewort erhalten hat**. Mit den für die Speicherung benötigten 14 Bit ist man wesentlich effizienter, als wenn man ERDBEERE mit ASCII-Codierung gespeichert hätte, was einem Speicherbedarf von 8x8Bit=64Bit entsprochen hätte. Die Versuchung liegt nahe, das Alphabet gemäss Häufigkeitsverteilung (der Buchstabe E kommt in deutschen und englischen Texten am häufigsten vor) in Huffman zu codieren und ASCII in Rente zu schicken. Was meinen sie dazu?

Beim Huffman-Algorithmus ist die Quellensymbolart nicht nur auf Buchstaben beschränkt. Huffman kommt z.B. auch bei der Bildkomprimierung (JPG/DCT) zur Anwednung.

Huffman-Baum: Verschiedene Varianten – dieselbe Effizienz:

Im folgenden Beispiel wurde das Wort **ERDBEERE** mit einem **N** zu **ERDBEEREN** ergänzt. Mehrere Lösungen sind nun möglich. Allerdings führen alle zur selben Codeeffizienz. (Das letzte Beispiel zeigt übrigens einen falschen Aufbau des binären Baums, was auch nicht zur selben Codeeffizienz wie bei den korrekten Beispielen führt)

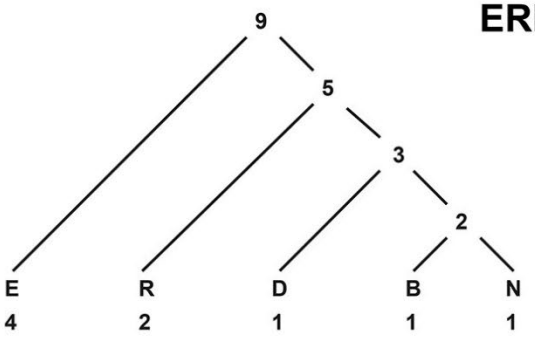
Da bei diesem Beispiel mehrere Codevarianten möglich sind, muss für die Huffman-Decodierung die gewählte Codetabelle mitgeliefert werden. Dies hat allerdings, im Vergleich zur ASCII-Variante, einen negativen Einfluss auf die Speicherbilanz. Der **Verwaltungsoverhead** ist im Vergleich zu den **Nutzdaten** zu hoch. Dies würde sich ändern, wenn der zu komprimierende Text markant länger wäre.

Im Folgenden ein weiteres Beispiel. Dem Wort ERDBEERE wurde einfach ein N angehängt: ERDBEEREN. Die folgende Grafik zeigt mehrere mögliche Huffman-Bäume, die alle dieselbe Effizienz aufweisen. Zum Schluss noch eine Baum-Variante, die falsch ist und demzufolge auch nicht zu derselben Komprimierung führt, wie die richtigen Beispiele.

Bild siehe nächste Seite!

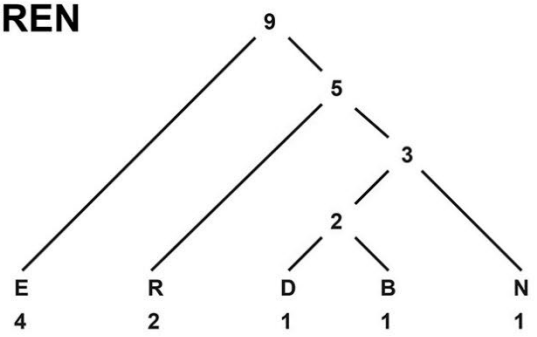
Verschiedene Baumvarianten mir derselben Code-Effizienz:

ERDBEEREN



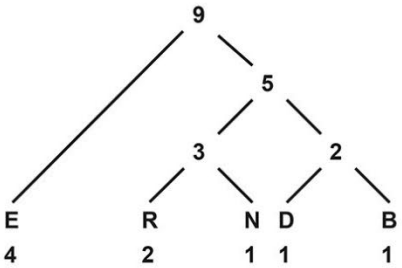
E=1
R=01
D=001
B=0001
N=0000

1. Variante:
1 01 001 0001 1 1 01 1 0000 = 19Bit
E R D B E E R E N



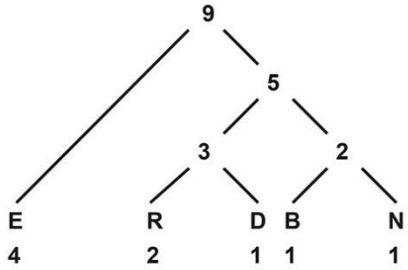
E=1
R=01
D=0011
B=0010
N=000

2. Variante:
1 01 0011 0010 1 1 01 1 000 = 19Bit
E R D B E E R E N



E=1
R=011
D=001
B=000
N=010

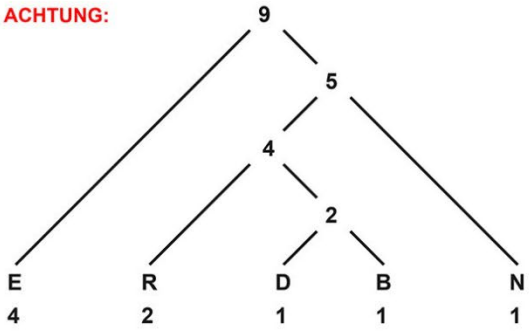
3. Variante:
1 011 001 000 1 1 011 1 010 = 19Bit
E R D B E E R E N



E=1
R=011
D=010
B=001
N=000

4. Variante:
1 011 010 001 1 1 011 1 000 = 19Bit
E R D B E E R E N

ACHTUNG:



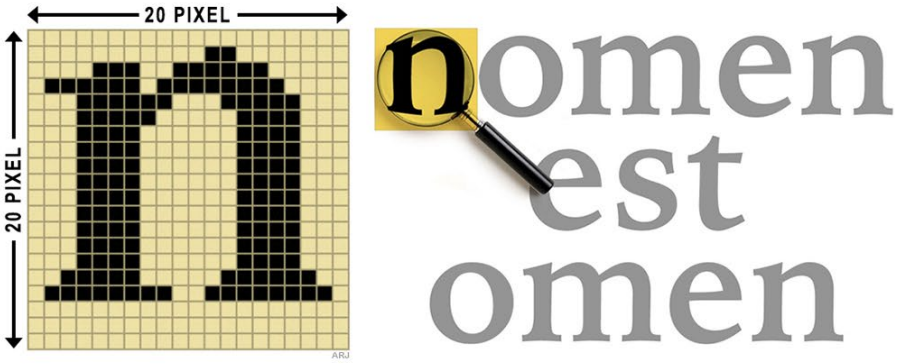
E=1
R=011
D=0101
B=0100
N=00

Keine Variante!
1 011 0101 0100 1 1 011 1 00 = 20Bit
E R D B E E R E N

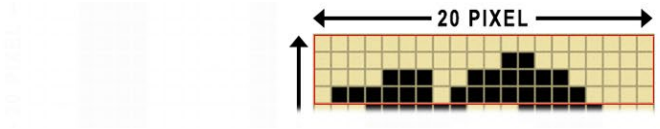
ARJ

RLC / RLE:

Neben VLC und Huffman gibt es mit RLC ein weiteres, wichtiges Verfahren, dass an vielen «Orten» anzutreffen ist. Mit **RLC**=**Run-Length-Coding** bzw. **RLE**=**Run-Length-Encoding** bedeutet, ist eine Lauflängenkodierung gemeint. Jede Sequenz von identischen Symbolen soll durch deren Anzahl und ggf. das Symbol ersetzt werden. Somit werden nur die Stellen markiert, an denen sich das Symbol in der Nachricht ändert. Effizient bei langen Wiederholungen.



RLC-Bit-Berechnung für die **<u>ersten vier Zeilen</u>**:



Der Speicherbedarf für das RLC-komprimierte Bild:
Ab erstem Pixel oben links: 31 x Weiss, 2 x Schwarz, 11 x Weiss, 3 x Schwarz, 2 x Weiss, 6 x Schwarz, 6 x Weiss, 6 x Schwarz, 1 x Weiss, 8 x Schwarz, 4 x Weiss

Total: 11 Zahlen oder Farbwechsel. Grösste Zahl: 31 (Benötigt 5 Bit)

Die Zahlen im Binärcode: 11111 - 00010 - 01011 - 00011 - 00010 - 00110 - 00110 - 00001 - 01000 - 00100

- Der Speicherbedarf für **RLC** gemäss obiger Berechnung: 11 x 5Bit = **55Bit** (Der Abstand mit Bindestrich wurde nur zwecks besserer Lesbarkeit hinzugefügt)
- Der Speicherbedarf im Vergleich dazu für das normale **Bitmap**: 4 Zeilen zu je 20 Pixel = **80Bit**

Optional: LZW, ein lexikalisches Verfahren (Lempel-Ziv-Welch-Algorithmus):

(Ob dieses Verfahren M114-Prüfungsrelevant ist, teilt ihnen die Lehrperson gerne auf Anfrage mit.)

Ein anderer Ansatz verfolgt der Lempel-Ziv-Welch-Algorithmus: Die Idee dahinter ist, wiederkehrende binäre Muster nicht erneut zu speichern, sondern eine Referenz darauf. Es wird eine Art Wörterbuch erstellt. LZW wird häufig bei der Datenreduktion von Grafikformaten (GIF, TIFF) verwendet.

ZW-Transformation am Textbeispiel ENTGEGENGENOMMEN erklärt

Das Wort «ENTGEGENGENOMMEN» soll verlustlos datenreduziert bzw. komprimiert werden. Die Werte 0 bis 255 sind den ASCII-Zeichen vorbehalten. Die Werte ab 256 sind Indexe, die auf einen Wörterbucheintrag verweisen:

		Wörterbuch-eintrag oder Buchstaben	ASCII 0..255 oder Index ab 256	Gespeicherter Wert und nächster Buchstaben in der Zeichenkette
Schritt	Zeichenkette	Gefunden	Gespeichert	Temporärer Wörterbucheintrag
1.	ENTGEGENGENOMMEN	E	E	EN → «256»
2.	ENTGEGENGENOMMEN	N	N	NT → «257»
3.	ENTGEGENGENOMMEN	T	T	TG → «258»
4.	ENTGEGENGENOMMEN	G	G	GE → «259»
5.	ENTGEGENGENOMMEN	E	E	EG → «260»
6.	ENTGEGENGENOMMEN	GE → «259»	«259»	GEN → «261»
7.	ENTGEGENGENOMMEN	N	N	NG → «262»
8.	ENTGEGENGENOMMEN	GEN → «261»	«261»	GENO → «263»
9.	ENTGEGENGENOMMEN	O	O	OM → «264»
10.	ENTGEGENGENOMMEN	M	M	MM → «265»
11.	ENTGEGENGENOMMEN	M	M	ME → «266»
12.	ENTGEGENGENOMMEN	EN → «256»	«256»	-

▲ Cursor-Position

LZW-Transformation: ENTGEGENGENOMMEN wird zu ENTGE259N261OMM256

Und die **LZW-Rücktransformation**:

		Zeichenkette, bestehend aus ASCII 0..255 oder Index ab 256	Erster Buchstabe in der Ausgabe	Buchstabe oder Wörterbuch	Vorherige Ausgabe und aktuelles Zeichen
Schritt	Übermittelte Zeichenkette	Aktuelles Zeichen	Ausgabe	Temporärer Wörterbucheintrag
1.	ENTGE«259»N«261»OMM«256»	E	E	-
2.	ENTGE«259»N«261»OMM«256»	N	N	EN → «256»
3.	ENTGE«259»N«261»OMM«256»	T	T	NT → «257»
4.	ENTGE«259»N«261»OMM«256»	G	G	TG → «258»
5.	ENTGE«259»N«261»OMM«256»	E	E	GE → «259»
6.	ENTGE«259»N«261»OMM«256»	G	GE	EG → «260»
7.	ENTGE«259»N«261»OMM«256»	N	N	GEN → «261»
8.	ENTGE«259»N«261»OMM«256»	G	GEN	NG → «262»
9.	ENTGE«259»N«261»OMM«256»	O	O	GENO → «263»
10.	ENTGE«259»N«261»OMM«256»	M	M	OM → «264»
11.	ENTGE«259»N«261»OMM«256»	M	M	MM → «265»
12.	ENTGE«259»N«261»OMM«256»	E	EN	ME → «266»

▲ Cursor-Position

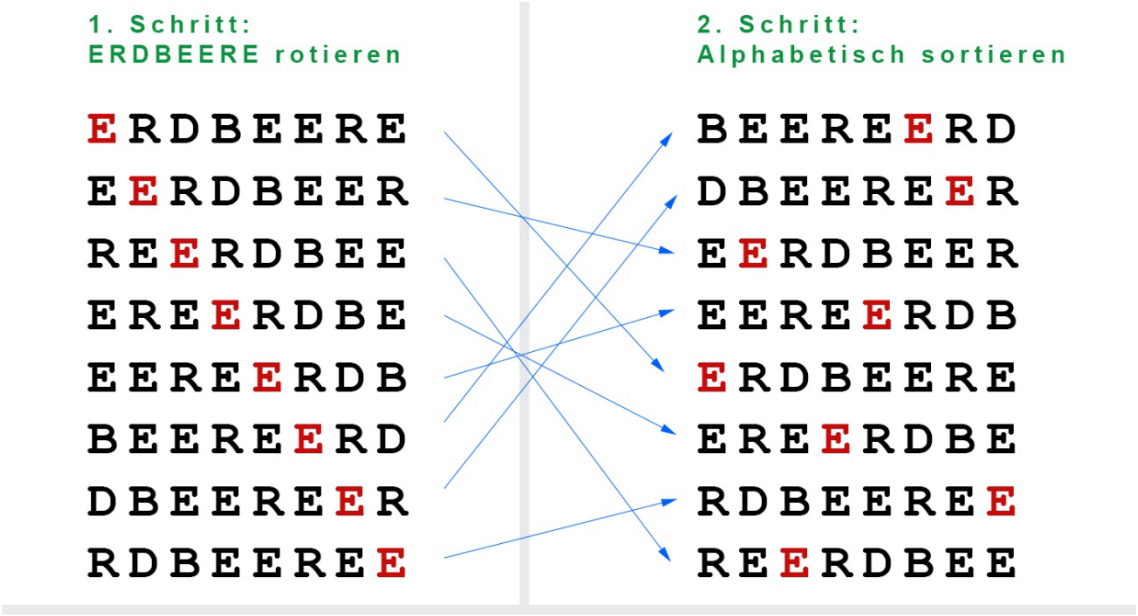
LZW-Rücktransformation: ENTGE259N261OMM256 wird zu ENTGEGENGENOMMEN

Optional: BWT (Burrows-Wheeler-Transformation):

(Ob dieses Verfahren M114-Prüfungsrelevant ist, teilt ihnen die Lehrperson gerne auf Anfrage mit.)

Neben RLC, Huffman und LZW ist BWT eine weitere Art, wie Daten verlustlos komprimiert werden können. BWT ist allerdings kein Algorithmus, der Daten direkt komprimiert. Vielmehr besteht seine Aufgabe darin, das Datenmaterial für eine anschliessende effektive Datenreduktion mit z.B. RLC vorzubereiten.

(Hinweis: BWT führt nicht in allen Fällen zu einer vorteilhaften Zusammenführung von gleichen Buchstaben. Ausserdem macht BWT bei längeren Textpassagen kein Sinn bzw. wäre mit viel Aufwand verbunden.)



3. Schritt:
Letzte Spalte und Position
des Originals werden übermittelt

1 : BEERERD
2 : DBEERER
3 : EERDBEER
4 : EERERDB
5 : ERDBEERE
6 : ERERDBE
7 : RDBEERE
8 : REERDBEE

Die eigentliche Datenreduktion erfolgt mit RLC

DRRBEEEE5 oder D2RB4E5

ARJ

Und die **BW-Rücktransformation**:

Übermittelte Zeichenfolge ...

CHAR	D	2	R		B	4	E		5

Expandierte Zeichenfolge ...

CHAR	D	R	R	B	E	E	E	E
POS	1	2	3	4	5	6	7	8

Alphabetisch sortiert... (POS wird zu NEXT / Startzeichen an POS 5)

POS	1	2	3	4	5	6	7	8
CHAR	B	D	E	E	E	E	R	R
NEXT	4	1	5	6	7	8	2	3

Originaltext erstellen... (NEXT zeigt auf den nächsten Buchstaben)

An POS 5 steht der erster Buchstabe «E». Der nächste Buchstabe steht an POS 7...

E

An POS 7 steht der zweite Buchstabe «R». Der nächste Buchstabe steht an POS 2...

E R

An POS 2 steht der dritte Buchstabe «D». Der nächste Buchstabe steht an POS 1...

E R D

An POS 1 steht der vierte Buchstabe «B». Der nächste Buchstabe steht an POS 4...

E R D B

An POS 4 steht der fünfte Buchstabe «E». Der nächste Buchstabe steht an POS 6...

E R D B E

An POS 6 steht der sechste Buchstabe «E». Der nächste Buchstabe steht an POS 8...

E R D B E E

An POS 8 steht der siebte Buchstabe «R». Der nächste Buchstabe steht an POS 3...

E R D B E E R

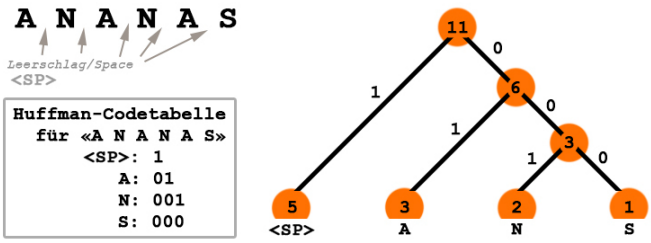
An POS 3 steht der achte und letzte Buchstabe «E».

E R D B E E R E

ARJ

Kombination von Huffman und RLC:

Heutzutage kombiniert man die Verfahren wie z.B. in DCT bei der JPG-Bildkomprimierung. Dazu ein Beispiel für das Wort **ANANAS**, das zuerst **Huffman** und danach **RLC**-komprimiert wird:




```
A    <SP>    N    <SP>    A    <SP>    N    <SP>    A    <SP>    S
01     1    001    1     01     1    001    1     01     1    000      Total: 20 Bit
                                                                    (Ohne Codetabelle!)
```

```
ASCII-Codetabelle
 <SP>: 00100000
    A: 01000001
    N: 01001110
    S: 01010011
```

```
     A       <SP>      N       <SP>      A       <SP>
01000001 00100000 01001110 00100000 01000001 00100000

     N       <SP>      A       <SP>      S
01001110 00100000 01000001 00100000 01010011   Total: 88 Bit
```

Kann man mit RLC noch etwas Platz sparen?

```
01000001001000000100111000100000010000010010000001001110001000000100000100100000001010011
Maximal 6 Nullen hintereinander -> 3 Bit zum Zählen der Nullen
Maximal 3 Einsen hintereinander -> 2 Bit zum Zählen der Einsen
Nullen und Einsen wechseln sich gegenseitig ab!

  0  1 00000  1  00  1 000000 1  00 111 000  1 000000  1 00000  1  00  1 000000 1   00 111 ...
001 01   101 01 010 01 110   01 010  11 011 01    110 01   101 01 010 01    110 01 010  11 ...

...000  1 000000  1 00000  1  00  1 000000  1   0  1  00 11
...011 01    110 01   110 01 010 01    110 01 001 01 010 10    Total: 90 Bit
                                                              (keine Ersparnis gegenüber ASCII!)
```

Huffman-Code für «A N A N A S» -> Weiter komprimieren mit RLC

```
01 1 001 1 01 1 001 1 01 1 000 ->     011001101100110110000
Je 2 Bit zum Zählen der Nullen bzw. Einsen
0  11 00 11 0  11 00 11 0  11 000  = Total: 20 Bit
01 11 11 11 01 11 11 11 01 11  11  = Total: 22 Bit
                                     (keine Ersparnis gegenüber Huffman!)

Neuer Ansatz: Lexikon. Abwechslungsweise 2Bit -> 1Bit etc.
0   -> 00
00  -> 01       0  11 00 11 0  11 00 11 0  11 000  = Total: 20 Bit
000 -> 10       00 1  01 1  00 1  01 1  00 1  10   = Total: 17 Bit
11  -> 1                                            (ohne Lexikon!)
```