Überblick zu Kapitel 2 von „Einführung in die Informationstheorie"

Anwendung findet die Shannonsche Informationstheorie zum Beispiel bei der **Quellencodierung** von digitalen (also wert– und zeitdiskreten) Nachrichtenquellen. Man spricht in diesem Zusammenhang auch von **Datenkomprimierung**. Dabei wird versucht, die Redundanz natürlicher Digitalquellen wie zum Beispiel Messdaten, Texte oder Sprach– und Bilddateien (nach Digitalisierung) durch Umcodierung zu vermindern, um diese effizienter speichern und übertragen zu können. Meist ist die Quellencodierung mit einer Änderung des Symbolumfangs verbunden. Im Folgenden sei die Ausgangsfolge stets binär.

Im Einzelnen werden behandelt:

- die unterschiedlichen Ziele von *Quellencodierung*, *Kanalcodierung* und *Leitungscodierung*,
- *verlustbehaftete* Codierverfahren für analoge Quellen, z.B. GIF, TIFF, JPEG, PNG, MP3,
- das *Quellencodierungstheorem*, das eine Grenze für die mittlere Codewortlänge angibt,
- die häufig angewandte *Datenkompression* nach *Lempel*, *Ziv* und *Welch*,
- der *Huffman–Code* als die bekannteste und effizienteste Form der Entropiecodierung,
- der *Shannon–Fano–Code* sowie die *arithmetische Codierung* – ebenfalls Entropiecodierer,
- die *Lauflängencodierung* sowie die *Burrows-Wheeler Transformation* (BWT).

Geeignete Literatur: [Abe03b] – [AM90] – [BCK02b] – [Bla87] – [CT06] – [Fan61] – [For72] – [Gal68] – [Joh92b] – [Kra13] – [McE77] – [Meck09] – [Söd01]

Die grundlegende Theorie wird auf 41 Seiten dargelegt. Außerdem beinhaltet dieses erste Kapitel noch 67 Grafiken, 14 Aufgaben und sieben Zusatzaufgaben mit insgesamt 103 Teilaufgaben sowie fünf Interaktionsmodule:

- **Entropien von Nachrichtenquellen** (Grundlagen)
- **Einfluss einer Bandbegrenzung für Sprache und Musik** (zu Kapitel 2.1 – Größe 9.2 MB)
- **Qualität verschiedener Sprach–Codecs** (zu Kapitel 2.1 – Größe 9.2 MB)
- **Lempel–Ziv–Algorithmen** (zu Kapitel 2.2)
- **Shannon–Fano– und Huffman–Codierung** (zu Kapitel 2.3)

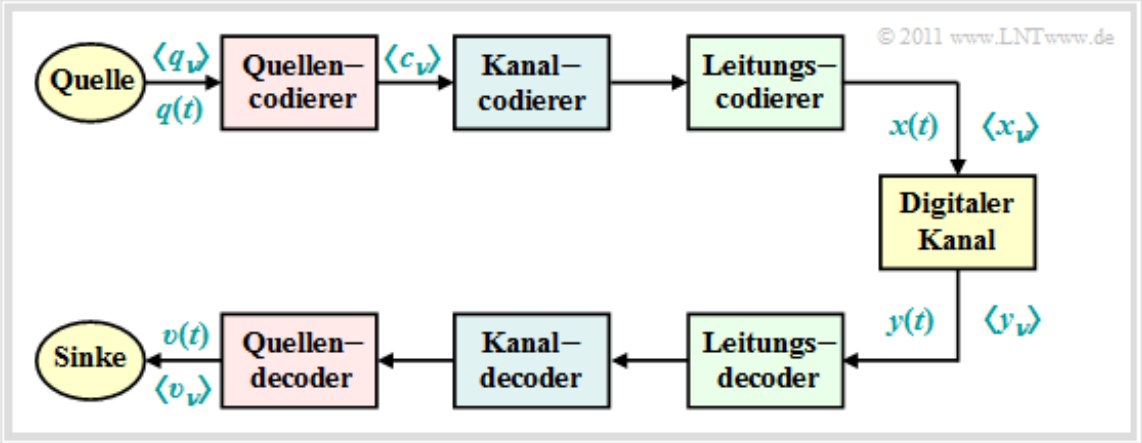
Weitere Informationen zu diesem Thema sowie Simulationsprogramme mit Grafikausgaben und Übungsaufgaben mit ausführlichen Musterlösungen finden Sie im Versuch „Wertdiskrete Informationstheorie" des Praktikums *Simulation digitaler Übertragungssysteme*, das seit Jahren von Prof. **Günter Söder** am Lehrstuhl für Nachrichtentechnik der TU München für Studierende der Elektro– und Informationstechnik angeboten wird:

Herunterladen des Windows–Programms „WDIT" (Zip–Version)

Herunterladen der dazugehörigen Texte (PDF–Datei)

Quellencodierung – Kanalcodierung – Leitungscodierung (1)

Wir betrachten für die Beschreibungen im Kapitel 2 das folgende digitale Übertragungsmodell:



Zu diesem Modell ist zu bemerken:

- Das Quellensignal $q(t)$ kann ebenso wie das Sinkensignal $v(t)$ sowohl analog als auch digital sein. Alle anderen Signale in diesem Blockschaltbild – auch die hier nicht explizit benannten – sind Digitalsignale.
- Insbesondere sind auch die Signale $x(t)$ und $y(t)$ am Eingang und Ausgang des *Digitalen Kanals* digital und können deshalb auch durch die Symbolfolgen $\langle x_\nu \rangle$ und $\langle y_\nu \rangle$ vollständig beschrieben werden.
- Der „Digitale Kanal" beinhaltet neben dem Übertragungsmedium und den Störungen (Rauschen) auch Komponenten des Senders (Modulator, Sendeimpulsformer, usw.) und des Empfängers (Demodulator, Empfangsfilter bzw. Detektor, Entscheider). Zur Modellierung des Digitalen Kanals sei auf das **Kapitel 5** im Buch „Digitalsignalübertragung" verwiesen.

Die Bildbeschreibung wird auf der nächsten Seite fortgesetzt.

Quellencodierung – Kanalcodierung – Leitungscodierung (2)

Wie aus dem **Blockschaltbild** der letzten Seite zu erkennen ist, unterscheidet man je nach Zielrichtung zwischen drei verschiedenen Arten von Codierung, jeweils realisiert durch den sendeseitigen Codierer (Coder) und den zugehörigen Decoder beim Empfänger:

- Die Aufgabe der **Quellencodierung** ist die Redundanzreduktion zur Datenkomprimierung, wie sie beispielsweise in der Bildcodierung Anwendung findet. Durch Ausnutzung statistischer Bindungen zwischen den einzelnen Punkten eines Bildes bzw. zwischen den Helligkeitswerten eines Punktes zu verschiedenen Zeiten (bei Bewegtbildsequenzen) können Verfahren entwickelt werden, die bei nahezu gleicher Bildqualität zu einer merklichen Verminderung der Datenmenge (gemessen in Bit oder Byte) führen. Ein einfaches Beispiel hierfür ist die differentielle Pulscodemodulation (DPCM).
- Bei der **Kanalcodierung** erzielt man demgegenüber dadurch eine merkliche Verbesserung des Übertragungsverhaltens, dass eine beim Sender gezielt hinzugefügte Redundanz empfangsseitig zur Erkennung und Korrektur von Übertragungsfehlern genutzt wird. Solche Codes, deren wichtigste Vertreter Blockcodes, Faltungscodes und Turbocodes sind, haben besonders bei stark gestörten Kanälen eine große Bedeutung. Je größer die relative Redundanz des codierten Signals ist, desto besser sind die Korrektureigenschaften des Codes, allerdings bei verringerter Nutzdatenrate.
- Eine **Leitungscodierung** – häufig auch als Übertragungscodierung bezeichnet – verwendet man, um das Sendesignal durch eine Umcodierung der Quellensymbole an die spektralen Eigenschaften von Kanal und Empfangseinrichtungen anzupassen. Beispielsweise muss bei einem Übertragungskanal, über den kein Gleichsignal übertragen werden kann – für den also $H_K(f = 0) = 0$ gilt – durch Übertragungscodierung sichergestellt werden, dass die Codesymbolfolge keine langen Folgen gleicher Polarität beinhaltet.

Im Mittelpunkt des vorliegenden Kapitels steht die **verlustfreie Quellencodierung**, die ausgehend von der Quellensymbolfolge $\langle q_\nu \rangle$ eine datenkomprimierte Codesymbolfolge $\langle c_\nu \rangle$ generiert, basierend auf den Ergebnissen der Informationstheorie.

Der Kanalcodierung ist in unserem Tutorial ein eigenes Buch mit folgendem **Inhalt** gewidmet. Die Leitungscodierung wird in **Kapitel 2** des Buches „Digitalsignalübertragung” eingehend behandelt.

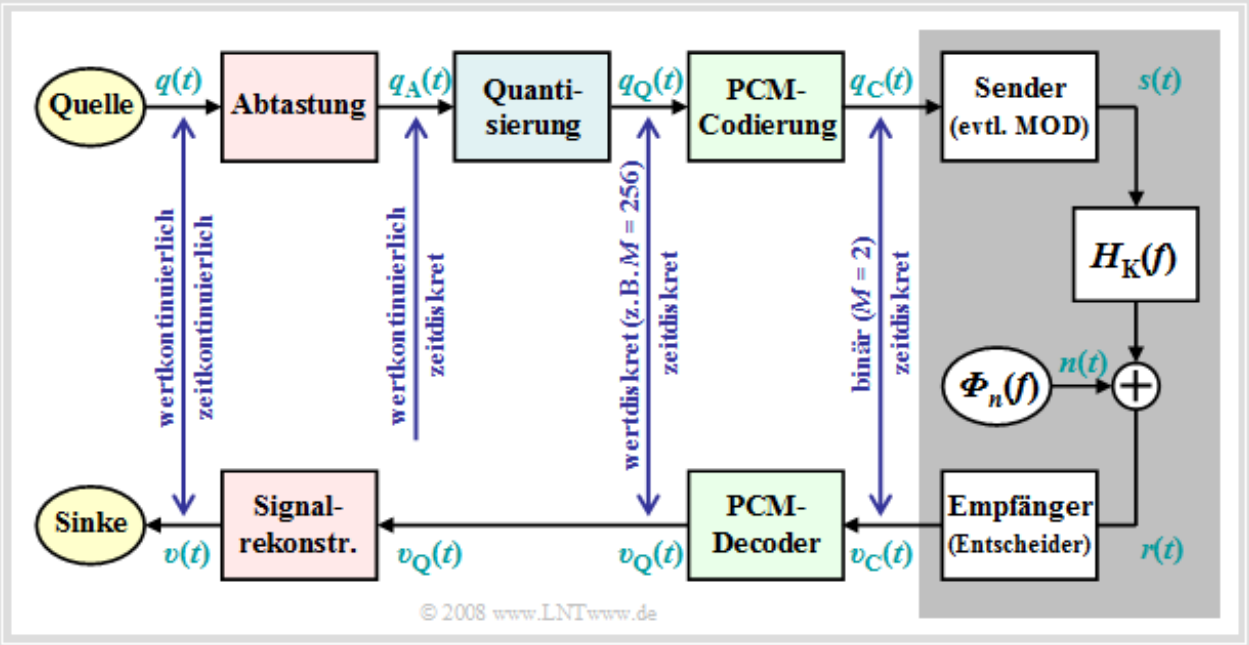
Anmerkung: Wir verwenden hier einheitlich „ν” als Laufvariable einer Symbolfolge. Eigentlich müssten für $\langle q_\nu \rangle$, $\langle c_\nu \rangle$ und $\langle x_\nu \rangle$ unterschiedliche Indizes verwendet werden, wenn die Raten nicht übereinstimmen.

Verlustbehaftete Quellencodierung (1)

Ein erstes Beispiel für Quellencodierung ist die 1938 erfundene **Pulscodemodulation** (PCM), die aus einem analogen Quellensignal $q(t)$ durch

- Abtastung
- Quantisierung
- PCM–Codierung

die Codesymbolfolge $\langle c_\nu \rangle$ extrahiert. Wegen der erforderlichen Bandbegrenzung und der Quantisierung ist diese Umformung jedoch stets verlustbehaftet. Das bedeutet, dass die codierte Folge $\langle c_\nu \rangle$ nicht die gesamte Information des Quellensignals $q(t)$ beinhaltet, und dass sich das Sinkensignal $v(t)$ grundsätzlich von $q(t)$ unterscheidet. Meist ist die Abweichung allerdings nicht sehr groß.



Die Grafik verdeutlicht das PCM–Prinzip. Die zugehörige Bildbeschreibung findet man auf den ersten Seiten von **Kapitel 4.1** im Buch „Modulationsverfahren".

Beispiel: Wird ein Sprachsignal spektral auf die Bandbreite $B = 4$ kHz $\Rightarrow$ Abtastrate $f_A = 8$ kHz begrenzt, so ergibt sich bei Quantisierung mit 13 Bit $\Rightarrow$ Quantisierungsstufenzahl $M = 2^{13} = 8192$ ein binärer Datenstrom der Datenrate $R = 104$ kbit/s. Die Daten entstammen der **GSM–Spezifikation.**

Der Quantisierungsrauschabstand beträgt dann $20 \cdot \lg M \approx 78$ dB. Bei Quantisierung mit 16 Bit würde sich dieser auf etwa 96 dB erhöhen, aber gleichzeizig steigt dadurch die erforderliche Datenrate auf 128 kbit/s. Die Auswirkungen der Bandbegrenzung auf ein Sprachsignal bzw. Musiksignal können Sie sich mit dem folgenden Interaktionsmodul verdeutlichen:

Einfluss einer Bandbegrenzung bei Sprache und Musik

Verlustbehaftete Quellencodierung (2)

Der Standard **ISDN** (*Integrated Services Digital Network*) für Telefonie über Zweidrahtleitung basiert auf dem PCM–Prinzip, wobei jedem Teilnehmer zwei B–Kanäle (*Bearer Channels*) mit je 64 kbit/s $\Rightarrow\ M = 2^8 = 256$ und ein D–Kanal (*Data Channel*) mit 16 kbit/s zur Verfügung gestellt wird. Die Nettodatenrate beträgt somit 144 kbit/s. Unter Berücksichtigung der Kanalcodierung und der Steuerbits (aus organisatorischen Gründen erforderlich) kommt man auf die ISDN–Bruttodatenrate von 192 kbit/s.

Im Mobilfunk können sehr große Datenraten oft (noch) nicht bewältigt werden. Hier wurden in den 1990er–Jahren Sprachcodierverfahren entwickelt, die zu einer Datenkomprimierung um den Faktor 8 und mehr führen. Zu erwähnen sind aus heutiger Sicht:

- der **Enhanced Full–Rate Codec** (EFR), der pro Sprachrahmen von 20 ms genau 244 Bit extrahiert (Datenrate: 12.2 kbit/s); erreicht wird diese Datenkomprimierung um mehr als den Faktor 8 durch die Aneinanderreihung mehrerer Verfahren: *Linear Predictive Coding* (LPC, Kurzzeitprädiktion), *Long Term Prediction* (LTP, Langzeitprädiktion) und *Regular Pulse Excitation* (RPE);
- der **Adaptive Multi–Rate Codec** (AMR), der auf **ACELP** (*Algebraic Code Excited Linear Prediction*) basiert und mehrere Modi zwischen 12.2 kbit/s (EFR) und 4.75 kbit/s bereit stellt, so dass bei schlechterer Kanalqualität eine verbesserte Kanalcodierung eingesetzt werden kann;
- der **Wideband–AMR** (WB–AMR) mit neun Modi zwischen 6.6 kbit/s und 23.85 kbit/s. Dieser wird bei UMTS eingesetzt und ist für breitbandigere Signale zwischen 200 Hz und 7 kHz geeignet. Die Abtastung erfolgt mit 16 kHz, die Quantisierung mit 4 Bit.

Das Audio–Interaktionsmodul **Qualität verschiedener Sprach–Codecs** vergleicht diese Codecs.

Zur Digitalisierung analoger Quellensignale wie Sprache, Musik oder Bilder können nur verlustbehaftete Quellencodierverfahren verwendet werden. Bereits die Speicherung eines Fotos im BMP–Format ist aufgrund von Abtastung, Quantisierung und der endlichen Farbtiefe stets mit einem Informationsverlust verbunden.

Daneben gibt es aber auch eine Vielzahl von Kompressionsverfahren für Bilder, die zu deutlich kleineren Bilddateien als „BMP" führen, zum Beispiel:

- **GIF** (*Graphics Interchange Format*), 1987 von Steve Wilhite entwickelt.
- **JPEG** – ein Format, das 1992 von der *Joint Photographie Experts Group* vorgestellt wurde und heute der Standard für Digitalkameras ist. Endung: „jpeg" bzw. „jpg".
- **TIFF** (*Tagged Image File Format*), um 1990 von Aldus Corp. (jetzt Adobe) und Microsoft entwickelt, ist noch heute der Quasi–Standard für druckreife Bilder höchster Qualität.
- **PNG** (*Portable Network Graphics*), 1995 von Thomas Boutell und Tom Lane entworfen als Ersatz für das durch Patentforderungen belastete GIF–Format; weniger komplex als TIFF.

Diese Kompressionsverfahren nutzen teilweise Vektorquantisierung zur Redundanzminderung korrelierter Bildpunkte, gleichzeitig die verlustlosen Kompressionsalgorithmen nach **Huffman** und **Lempel/Ziv,**

eventuell auch Transformationscodierungen basierend auf DFT (*Diskrete Fouriertransformation*) und DCT (*Diskrete Cosinustransformation*), danach Quantisierung und Übertragung im transformierten Bereich.

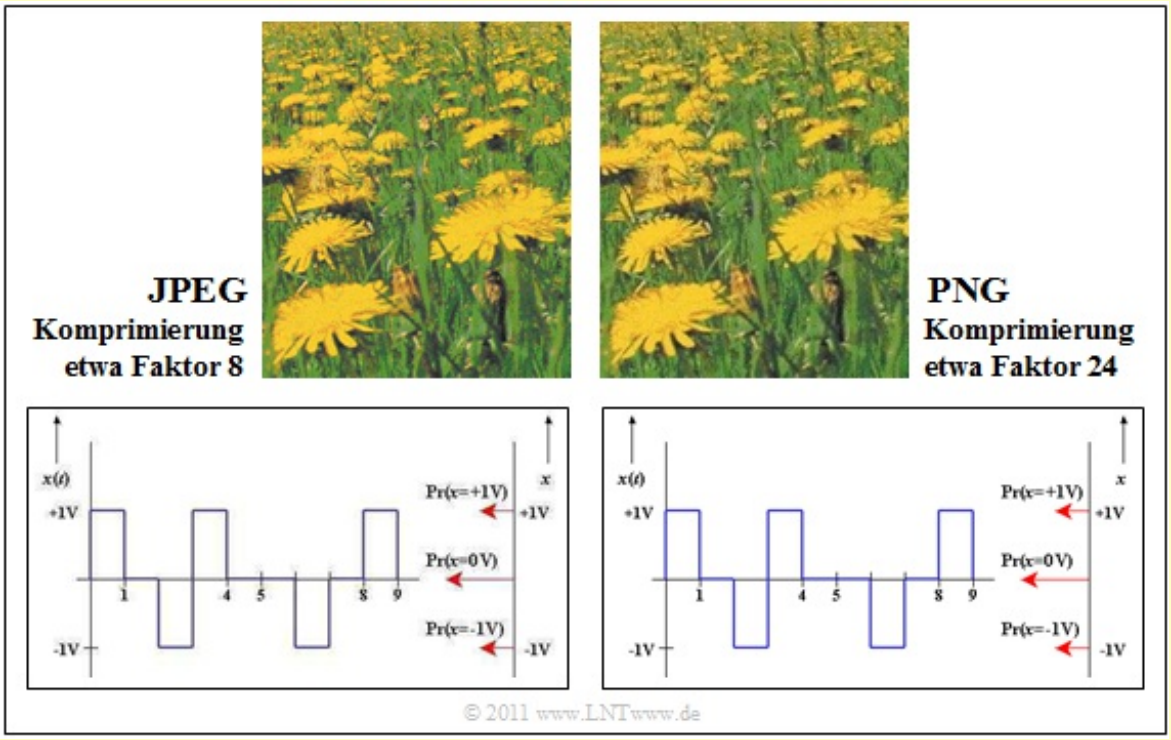
Verlustbehaftete Quellencodierung (3)

Wir vergleichen nun die Auswirkungen von

- JPEG (mit Komprimierungsfaktor 8) und
- PNG (mit Komprimierungsfaktor 24)

auf die subjektive Qualität von Fotos und Grafiken.

Beispiel: Im oberen Teil der folgenden Grafik sehen Sie zwei Komprimierungen eines Fotos. Das Format JPEG (linke Darstellung) ermöglicht gegenüber der pixelweisen Abspeicherung einen Komprimierungsfaktor von 8 bis 15 bei (nahezu) verlustfreier Komprimierung. Selbst mit dem Faktor 35 kann das Ergebnis noch als „gut" bezeichnet werden.



Das rechts dargestellte Bild wurde mit PNG komprimiert. Die Qualität ist vergleichbar mit dem linken JPEG–Bild, obwohl die Komprimierung um etwa den Faktor 3 stärker ist. Dagegen erzielt PNG ein schlechteres Komprimierungsergebnis als JPEG, wenn das Foto sehr viele Farbstufungen enthält. Bei den meisten Digitalkameras für den Consumer–Bereich ist JPEG das voreingestellte Speicherformat.

Auch bei Strichzeichnungen mit Beschriftungen ist PNG besser geeignet als JPEG (untere Bilder). Die Qualität der JPEG–Komprimierung (links) ist deutlich schlechter als das PNG–Resultat, obwohl die resultierende Dateigröße etwa dreimal so groß ist. Insbesondere Schriften wirken „verwaschen".

Anmerkung: Aufgrund technischer Einschränkungen bei LNTwww mussten alle Grafiken als PNG gespeichert werden. In obiger Grafik bedeutet also „JPEG" die PNG–Konvertierung einer zuvor mit JPEG komprimierten Datei. Der damit zusammenhängende Verlust ist jedoch vernachlässigbar.

MPEG–2 Audio Layer III – kurz MP3

Das heute (2015) am weitesten verbreitete Kompressionsverfahren für Audiodateien ist MP3. Entwickelt wurde dieses Format ab 1982 am Fraunhofer–Institut für Integrierte Schaltungen (IIS) in Erlangen unter der Federführung von Prof. Hans–Georg Musmann in Zusammenarbeit mit der Friedrich–Alexander–Universität Erlangen–Nürnberg und den AT&T Bell Labs. Auch andere Institutionen machen diesbezügliche Patentansprüche geltend, so dass seit 1998 zu verschiedene Klagen gab, die nach Kenntnis der Autoren noch nicht endgültig abgeschlossen sind.

Im Folgenden werden einige Maßnahmen genannt, die bei MP3 genutzt werden, um die Datenmenge gegenüber der Raw–Version im WAV–Format zu reduzieren. Die Zusammenstellung ist nicht vollständig. Eine umfassende Darstellung findet man zum Beispiel in **WIKIPEDIA.**

- Das Audio–Kompressionsverfahren MP3 nutzt unter anderem auch psychoakustische Effekte der Wahrnehmung aus. So kann der Mensch zwei Töne erst ab einem gewissen Mindestunterschied der Tonhöhe voneinander unterscheiden. Man spricht von so genannten „Maskierungseffekten”.
- Die Maskierungseffekte ausnutzend werden bei MP3 Signalanteile, die für den Höreindruck minderwichtig sind, mit weniger Bit (verringerte Genauigkeit) gespeichert. Ein dominanter Ton bei 4 kHz kann beispielsweise dazu führen, dass benachbarte Frequenzen bis zu 11 kHz für das momentane Hörempfinden nur eine untergeordnete Bedeutung besitzen.
- Die größte Ersparnis der MP3–Codierung liegt aber daran, dass die Töne mit gerade so vielen Bits abgespeichert werden, dass das dadurch entstehende **Quantisierungsrauschen** noch maskiert wird und nicht hörbar ist.
- Weitere MP3–Kompressionsmechanismen sind die Ausnutzung der Korrelationen zwischen den beiden Kanälen eines Stereosignals durch Differenzbildung sowie die **Huffman–Codierung** des resultierenden Datenstroms. Beide Maßnahmen sind verlustlos.

Nachteil der MP3–Codierung ist, dass bei starker Kompression auch „wichtige” Frequenzanteile von der Kompression erfasst werden und es dadurch zu hörbaren Fehlern kommt. Ferner ist es störend, dass aufgrund der blockweisen Anwendung des MP3–Verfahrens am Ende einer Datei Lücken entstehen können. Abhilfe schafft die Verwendung des so genannten LAME–Coders – ein *Open–Source–Project* – und eines entsprechenden Abspielprogramms.

Voraussetzungen für Kapitel 2

Im Folgenden betrachten wir ausschließlich verlustlose Quellencodierverfahren und gehen dabei von folgenden Annahmen aus:

- Die digitale Quelle besitze den Symbolumfang M. Für die einzelnen Quellensymbole der Folge $\langle q_\nu \rangle$ gelte mit dem Symbolvorrat $\{q_\mu\}$:

$$q_\nu \in \{q_\mu\}, \quad \mu = 1, \ldots, M.$$

- Die einzelnen Folgenelemente q_ν können statistisch unabhängig sein oder auch statistische Bindungen aufweisen. Zunächst betrachten wir Nachrichtenquellen **ohne Gedächtnis**, die durch die Symbolwahrscheinlichkeiten vollständig charakterisiert sind; zum Beispiel:

$M = 4$: $q_\mu \in \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$, mit den Wahrscheinlichkeiten p_A, p_B, p_C, p_D,
$M = 8$: $q_\mu \in \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}, \mathbf{G}, \mathbf{H}\}$, Wahrscheinlichkeiten $p_A, \ldots, p_H$.

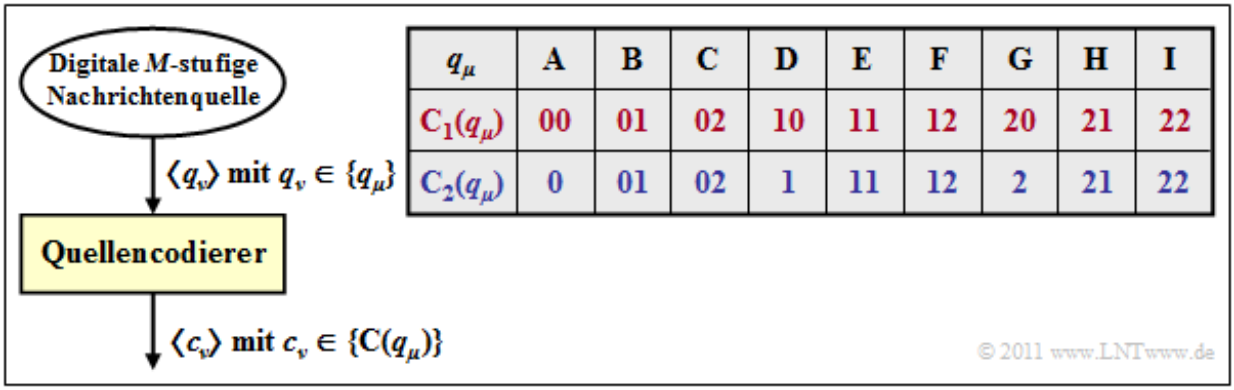
- Der Quellencodierer ersetzt das Quellensymbol q_μ durch das Codewort $C(q_\mu)$, bestehend aus L_μ Codesymbolen eines neuen Alphabets $\{0, 1, \ldots, D-1\}$ mit dem Symbolumfang D. Damit ergibt sich für die **mittlere Codewortlänge**:

$$L_M = \sum_{\mu=1}^{M} p_\mu \cdot L_\mu, \quad \text{mit} \quad p_\mu = \Pr(q_\mu).$$

Beispiel: Wir betrachten zwei verschiedene Quellencodierungen, jeweils mit den Parametern $M = 9$ und $D = 3$. Bei der ersten Codierung $C_1(q_\mu)$ entsprechend Zeile 2 (rote Darstellung) wird jedes Quellensymbol q_μ durch zwei Ternärsymbole (0, 1 oder 2) ersetzt. Beispielsweise gilt die Zuordnung:

A C F B I G ⇒ 00 02 12 01 22 20

Bei dieser Codierung haben alle Codeworte $C_1(q_\mu)$ mit $1 \le \mu \le 9$ die gleiche Länge $L_\mu = 2$. Damit ist auch die mittlere Codewortlänge $L_M = 2$.



q_μ	A	B	C	D	E	F	G	H	I
$C_1(q_\mu)$	00	01	02	10	11	12	20	21	22
$C_2(q_\mu)$	0	01	02	1	11	12	2	21	22

Dagegen gilt beim zweiten, dem blauen Quellencodierer $L_\mu \in \{1, 2\}$ und dementsprechend wird die mittlere Codewortlänge kleiner sein als zwei Codesymbole pro Quellensymbol. Hier gilt die Zuordnung:

A C F B I G ⇒ 0 02 12 01 22 2.

Es ist offensichtlich, dass diese zweite Codesymbolfolge nicht eindeutig decodiert werden kann.

Kraftsche Ungleichung – Präfixfreie Codes

Codes zur Komprimierung einer gedächtnislosen wertdiskreten Quelle zeichnen sich dadurch aus, dass die einzelnen Symbole durch verschieden lange Codesymbolfolgen dargestellt werden:

$$L_\mu \neq \text{const.} \quad (\mu = 1, \ldots, M)\,.$$

Nur dann ist es möglich,

- dass die **mittlere Codewortlänge minimal** wird,
- falls die **Quellensymbole nicht gleichwahrscheinlich** sind.

Um eine eindeutige Decodierung zu ermöglichen, muss der Code zudem „präfixfrei" sein.

Definition: Die Eigenschaft **präfixfrei** sagt aus, dass kein Codewort der Präfix (der Beginn) eines längeren Codewortes sein darf. Ein solcher präfixfreier Code ist sofort decodierbar.

Der zweite (blaue) Code im **Beispiel** auf der letzten Seite ist nicht präfixfrei. Beispielsweise könnte die Codesymbolfolge „01" vom Decoder als **AD** interpretiert werden, aber ebenso als **B**. Dagegen ist der rote Code präfixfrei, wobei hier die Präfixfreiheit wegen L_μ = const. nicht unbedingt erforderlich wäre.

Die notwendige Bedingung für die Existenz eines präfixfreien Codes wurde von Leon Kraft in seiner Master Thesis 1949 am *Massachusetts Institute of Technology* (MIT) angegeben **[Kra49]:**

$$\sum_{\mu=1}^{M} D^{-L_\mu} \leq 1\,.$$

Beispiel 1: Überprüft man den zweiten (blauen) Code des betrachteten Beispiels mit $M = 9$ und $D = 3$, so erhält man:

$$3 \cdot 3^{-1} + 6 \cdot 3^{-2} = 1.667 > 1\,.$$

Daraus ist ersichtlich, dass dieser Code nicht präfixfrei sein kann.

Beispiel 2: Betrachten wir den binären Code

$$\mathbf{A} \Rightarrow 0, \quad \mathbf{B} \Rightarrow 00, \quad \mathbf{C} \Rightarrow 11,$$

so ist dieser offensichtlich nicht präfixfrei. Die Gleichung

$$1 \cdot 2^{-1} + 2 \cdot 2^{-2} = 1$$

sagt also keinesfalls aus, dass dieser Code tatsächlich präfixfrei ist, sondern es bedeutet lediglich, dass es einen präfixfreien Code mit gleicher Längenverteilung gibt, zum Beispiel

$$\mathbf{A} \Rightarrow 0, \quad \mathbf{B} \Rightarrow 10, \quad \mathbf{C} \Rightarrow 11.$$

Quellencodierungstheorem (1)

Wir betrachten eine redundante Nachrichtenquelle mit dem Symbolvorrat $\{q_\mu\}$, wobei die Laufvariable μ alle Werte zwischen 1 und dem Symbolumfang M annimmt. Die Quellenentropie H sei kleiner als der Nachrichtengehalt H_0.

Die Redundanz $H_0 - H$ geht entweder zurück

- auf nicht gleichwahrscheinliche Symbole $\Rightarrow$ $p_\mu \neq 1/M$, und/oder
- auf statistische Bindungen innerhalb der Folge $\langle q_\mu \rangle$.

Ein Quellencodierer ersetzt das Quellensymbol q_μ durch das binäre Codewort $C(q_\mu)$, bestehend aus L_μ Binärsymbolen (Nullen oder Einsen). Damit ergibt sich die mittlere Codewortlänge zu

$$L_{\rm M} = \sum_{\mu=1}^{M} p_\mu \cdot L_\mu \,, \quad \text{mit} \quad p_\mu = {\rm Pr}(q_\mu) \,.$$

Für die hier beschriebene Quellencodierungsaufgabe kann folgende Grenze angegeben werden:

Shannons Quellencodierungstheorem: Für die vollständige Rekonstruktion der gesendeten Zeichenfolge aus der Binärfolge ist es hinreichend, aber auch notwendig, dass man zur sendeseitigen Codierung im Mittel H Binärsymbole pro Quellensymbol verwendet. Das heißt, dass die mittlere Codewortlänge auf keinen Fall kleiner sein kann als die Entropie H der Quellensymbolfolge:

$$L_{\rm M} \geq H \,.$$

Berücksichtigt der Quellencodierer nur die unterschiedlichen Auftrittswahrscheinlichkeiten, nicht aber die inneren statistischen Bindungen, dann gilt $L_M \geq H_1$ $\Rightarrow$ **erste Entropienäherung.**

Beispiel 1: Bei einer Quaternärquelle mit den Symbolwahrscheinlichkeiten

$$p_{\rm A} = 2^{-1} \,, \;\; p_{\rm B} = 2^{-2} \,, \;\; p_{\rm C} = p_{\rm D} = 2^{-3} \quad \Rightarrow \quad H = H_1 = 1.75 \text{ bit/Quellensymbol}$$

ergibt sich in obiger Gleichung das Gleichheitszeichen $\Rightarrow$ $L_M = H$, wenn man zum Beispiel folgende Zuordnung wählt:

$$\mathbf{A} \;\Rightarrow\; 0, \;\; \mathbf{B} \;\Rightarrow\; 10, \;\; \mathbf{C} \;\Rightarrow\; 110, \;\; \mathbf{D} \;\Rightarrow\; 111 \,.$$

Dagegen ergibt sich mit der gleichen Zuordnung und

$$p_{\rm A} = 0.4 \,, \;\; p_{\rm B} = 0.3 \,, \;\; p_{\rm C} = 0.2 \,, \;\; p_{\rm D} = 0.1 \quad \Rightarrow \quad H = 1.845 \text{ bit/Quellensymbol}$$

die mittlere Codewortlänge

$$L_{\rm M} = 0.4 \cdot 1 + 0.3 \cdot 2 + 0.2 \cdot 3 + 0.1 \cdot 3 = 1.9 \text{ bit/Quellensymbol} \,.$$

Wegen der ungünstigen Symbolwahrscheinlichkeiten (keine Zweierpotenzen) ist hier $L_M > H$.

Es folgt ein zweites Beispiel, wobei die Quellensymbolfolge einen natürlichen Text beschreibt.

Quellencodierungstheorem (2)

Beispiel 2: Betrachten wir noch frühere Versuche der Quellencodierung für die Übertragung von Texten, wobei wir von den in der Tabelle angegebenen Buchstabenhäufigkeiten ausgehen. In der Literatur findet man eine Vielzahl unterschiedlicher Häufigkeiten, auch deshalb, weil verschiedene Autoren ihre Untersuchungen für verschiedene Sprachen durchführten. Meist beginnt die Liste aber mit dem Leerzeichen (Blank) und „E" und endet mit Buchstaben wie „X", „Y" und „Q".

Buchstabe	Blank	E	N	S	I	...	M	...	X	Q
Häufigkeit [%]	14.42	14.40	8.65	6.46	6.28		1.72		0.08	0.05
Bacon (1623) Baudot (1874)	00100	10000	00110	10100	01100		00111		10111	11101
Morse (1844)		·	— ·	· · ·	· ·		— —		— · · —	— — · —
Huffman (1952)	000	001	010	0110	0111		111010		1111111110	11111111110

Zu obiger Tabelle ist zu bemerken:

- Die Entropie dieses Alphabets mit $M = 27$ Zeichen wird $H \approx 4$ bit/Zeichen betragen. Wir haben das nicht nachgerechnet. Bacon hat aber schon 1623 einen Binärcode angegeben, bei dem jeder Buchstabe mit fünf Bit dargestellt wird: $L_M = 5$.
- Etwa 250 Jahre danach hat **Baudot** diesen Code übernommen, der später auch für die gesamte Telegrafie standardisiert wurde. Eine ihm wichtige Überlegung war, dass ein Code mit einheitlich fünf Binärzeichen pro Buchstabe für einen Feind schwerer zu dechiffrieren ist, da dieser aus der Häufigkeit des Auftretens keine Rückschlüsse auf das übertragene Zeichen ziehen kann.
- Die letzte Zeile gibt einen beispielhaften **Huffman–Code** für obige Häufigkeitsverteilung an. Wahrscheinliche Zeichen wie „E" oder „N" und auch das „Blank" werden mit nur drei Bit dargestellt, das seltene „Q" dagegen mit 11 Bit. Die mittlere Codewortlänge ist geringfügig größer als $H \Rightarrow L_M = H + \varepsilon$, wobei wir uns hier über das ε nicht auslassen wollen. Nur soviel: Es gibt keinen präfixfreien Code mit kleinerer mittlerer Wortlänge als den Huffman–Code.
- Auch **Samuel Morse** berücksichtigte bereits bei seinem Code für die Telegrafie in den 1830er Jahren die unterschiedlichen Häufigkeiten. Der Morse–Code eines jeden Zeichens besteht aus zwei bis vier Binärzeichen, die hier entsprechend der Anwendung mit Punkt („Kurz") und Strich („Lang") bezeichnet werden.
- Es ist offensichtlich, dass für den Morsecode $L_M < 4$ gelten wird. Dies hängt aber auch damit zusammen, dass dieser nicht präfixfrei ist. Zwischen jeder Kurz–Lang–Sequenz musste deshalb der Funker eine Pause einlegen, damit die Gegenstation das Funksignal auch entschlüsseln konnte.

Statische und dynamische Wörterbuchtechniken (1)

Viele Datenkomprimierungsverfahren verwenden Wörterbücher. Die Idee ist dabei die Folgende: Man konstruiere eine Liste der Zeichenmuster, die im Text vorkommen, und codiere diese Muster als Indizes der Liste. Besonders effizient ist diese Vorgehensweise, wenn sich bestimmte Muster im Text häufig wiederholen und dies bei der Codierung auch berücksichtigt wird. Hierbei unterscheidet man:

- Verfahren mit statischem Wörterbuch,
- Verfahren mit dynamischem Wörterbuch (Beschreibung auf der nächsten Seite).

Verfahren mit statischem Wörterbuch

Ein statisches Wörterbuch ist nur für ganz spezielle Anwendungen sinnvoll, zum Beispiel für eine Datei der folgenden Form:

Beispielsweise ergibt sich mit den Zuordnungen

$$\text{"0"} \mapsto 000000,\ \text{...},\ \text{"9"} \mapsto 001001,\ \text{"\_" (Blank)} \mapsto 001010,$$
$$\text{"." (Punkt)} \mapsto 001011,\ \text{"," (Komma)} \mapsto 001011,\ \text{"end-of-line} \mapsto 001101,$$
$$\text{"A"} \mapsto 100000,\ \text{...},\ \text{"E"} \mapsto 100100,\ \text{...},\ \text{"L"} \mapsto 101011,\ \text{"M"} \mapsto 101100,$$
$$\text{"O"} \mapsto 101110,\ \text{...},\ \text{"U"} \mapsto 110100,\ \text{"Name: \_"} \mapsto 010000,$$
$$\text{", \_Vorname: \_"} \mapsto 010001,\ \text{", \_Wohnort: \_"} \mapsto 010010,\ \text{...}$$

für die mit 6 Bit pro Zeichen binär–quellencodierte erste Zeile des obigen Textes:

$$010000\ 100000\ 100001\ 100100\ 101011 \ \Rightarrow\ (\text{Name: \_})\,(\text{A})\,(\text{B})\,(\text{E})\,(\text{L})$$
$$010001\ 101011\ 100100\ 101110 \ \Rightarrow\ (\text{, \_Vorname: \_})\,(\text{L})\,(\text{E})\,(\text{O})$$
$$010010\ 110100\ 101011\ 101100 \ \Rightarrow\ (\text{, \_Wohnort: \_})\,(\text{U})\,(\text{L})\,(\text{M})$$
$$001101 \ \Rightarrow\ (\text{end-of-line})$$

Bei dieser spezifischen Anwendung lässt sich die erste Zeile mit $14 \cdot 6 = 84$ Bit darstellen. Dagegen würde man bei herkömmlicher Binärcodierung $39 \cdot 7 = 273$ Bit benötigen (aufgrund der Kleinbuchstaben im Text reichen hier 6 Bit pro Zeichen nicht aus). Für den gesamten Text ergeben sich $103 \cdot 6 = 618$ Bit gegenüber $196 \cdot 7 = 1372$ Bit. Allerdings muss die Codetabelle auch dem Empfänger bekannt sein.

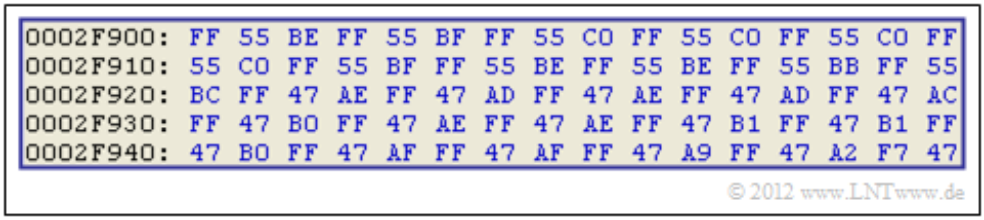
Statische und dynamische Wörterbuchtechniken (2)

Verfahren mit dynamischem Wörterbuch

Alle relevanten Komprimierungsverfahren arbeiten allerdings nicht mit statischem Wörterbuch, sondern mit dynamischen Wörterbüchern, die erst während der Codierung sukzessive entstehen:

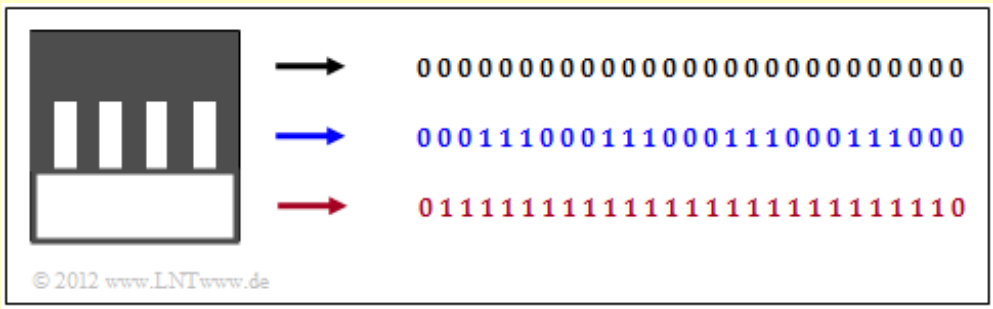
- Solche Verfahren sind flexibel einsetzbar und müssen nicht an die Anwendung adaptiert werden. Man spricht von *universellen Quellencodierverfahren*.
- Es genügt dann ein einziger Durchlauf, während bei Verfahren mit statischem Wörterbuch die Datei vor dem Codiervorgang erst analysiert werden muss.
- An der Sinke wird das dynamische Wörterbuch in gleicher Weise generiert wie bei der Quelle. Damit entfällt die Übertragung des Wörterbuchs.

Beispiel 1: Die Grafik zeigt einen kleinen Ausschnitt von 80 Byte einer **BMP–Datei** in Hexadezimaldarstellung. Es handelt sich um die unkomprimierte Darstellung eines natürlichen Bildes.



Man erkennt, dass in diesem kleinen Ausschnitt einer Landschaftsaufnahme die Bytes **FF**, **55** und **47** sehr häufig auftreten. Eine Datenkomprimierung ist deshalb erfolgversprechend. Da aber an anderen Stellen der 4 MByte–Datei oder bei anderem Bildinhalt andere Bytekombinationen dominieren, wäre hier die Verwendung eines statischen Wörterbuchs nicht zielführend.

Beispiel 2: Bei einer künstlich erzeugten Grafik – zum Beispiel bei einem Formular – könnte man dagegen durchaus mit einem statischen Wörterbuch arbeiten. Wir betrachten hier ein S/W–Bild mit 27 × 27 Pixeln, wobei die Zuordnung „Schwarz” ⇒ **0** und „Weiß” ⇒ **1** vereinbart wurde.



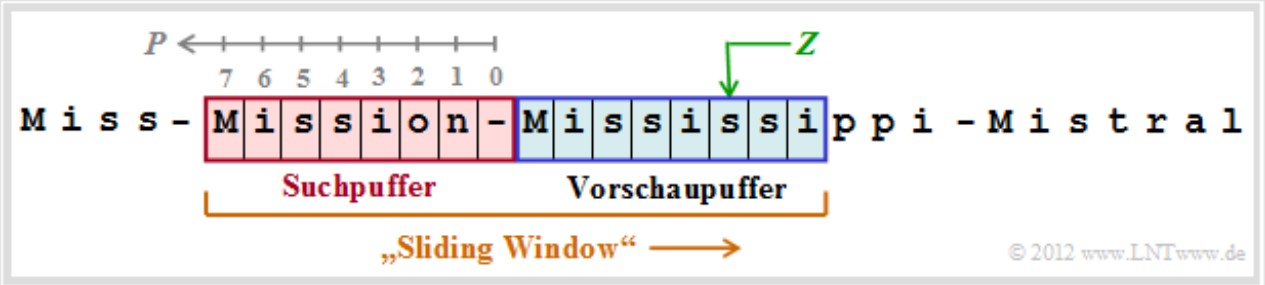
- Im oberen Bereich (schwarze Markierung) wird jede Zeile durch 27 Nullen beschrieben.
- In der Mitte (blaue Markierung) wechseln sich jeweils drei Nullen und drei Einsen ab.
- Unten (rote Markierung) werden pro Zeile 25 Einsen durch zwei Nullen begrenzt.

LZ77 – die Grundform der Lempel–Ziv–Algorithmen (1)

Die wichtigsten Verfahren zur Datenkomprimierung mit dynamischem Wörterbuch gehen auf **Abraham Lempel** und **Jacob Ziv** zurück. Die gesamte Lempel–Ziv–Familie (im Folgenden verwenden wir hierfür kurz: LZ–Verfahren) kann wie folgt charakterisiert werden:

- Lempel–Ziv–Verfahren nutzen die Tatsache, dass in einem Text oft ganze Wörter – oder zumindest Teile davon – mehrfach vorkommen. Man sammelt alle Wortfragmente, die man auch als *Phrasen* bezeichnet, in einem ausreichend großen Wörterbuch.
- Im Gegensatz zur vorher entwickelten **Entropiecodierung** (z.B. von Shannon und Huffman) ist hier nicht die Häufigkeit einzelner Zeichen oder Zeichenfolgen die Grundlage der Komprimierung, so dass die LZ–Verfahren auch ohne Kenntnis der Quellenstatistik angewendet werden können.
- LZ–Komprimierungsverfahren kommen dementsprechend mit einem einzigen Durchgang aus und auch der Quellensymbolumfang M und die Symbolmenge $\{q_\mu, \mu = 1, \text{ ... }, M\}$ muss nicht bekannt sein. Man spricht von **universeller Quellencodierung** (englisch: *Universal Source Coding*).

Wir betrachten zunächst den Lempel–Ziv–Algorithmus in seiner ursprünglichen Form aus dem Jahre 1977, bekannt unter der Bezeichnung **LZ77.** Dieser arbeitet mit einem Fenster, das sukzessive über den Text verschoben wird; man spricht auch von einem *Sliding Window*. Die Fenstergröße G ist dabei ein wichtiger Parameter, der das Komprimierungsergebnis entscheidend beeinflusst.



Die Grafik zeigt eine beispielhafte Belegung des *Sliding Windows*. Dieses ist unterteilt in

- den Vorschaupuffer (blaue Hinterlegung) und
- den Suchpuffer (rote Hinterlegung, mit Positionen $P = 0, \text{ ... }, 7 \;\Rightarrow\; G = 8$).

Der bearbeitete Text umfasst die vier Worte **Miss**, **Mission**, **Mississippi** und **Mistral**, jeweils getrennt durch einen Bindestrich. Zum betrachteten Zeitpunkt steht im Vorschaupuffer **Mississi**.

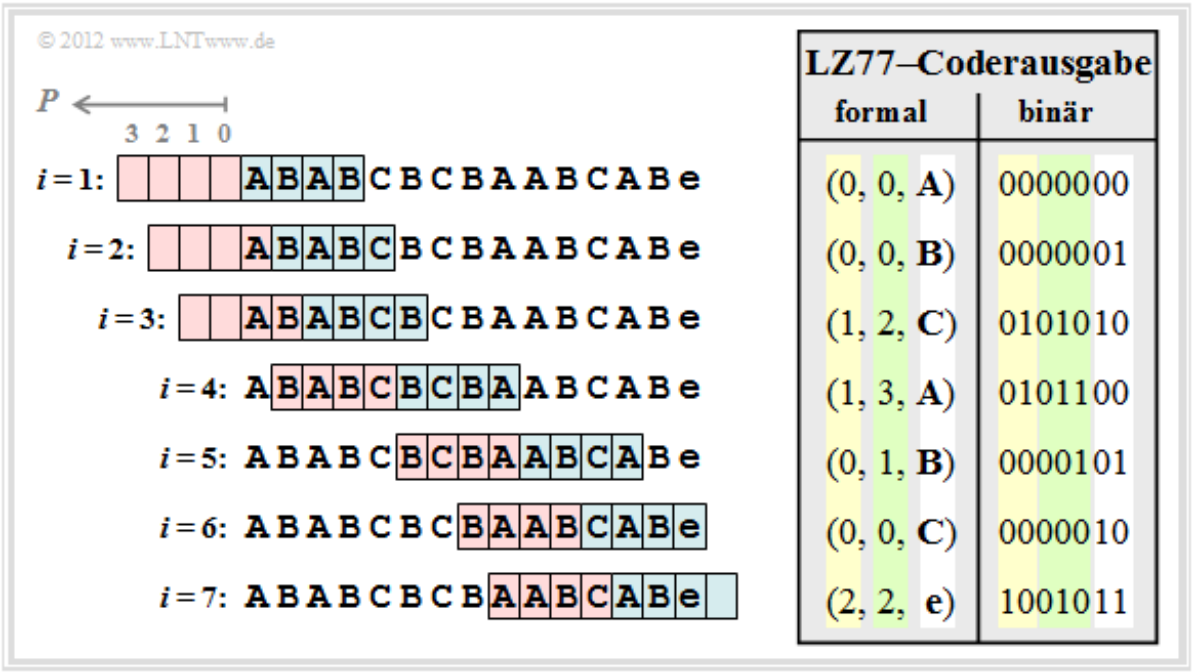
- Gesucht wird nun im Suchpuffer die beste Übereinstimmung $\Rightarrow$ die Zeichenfolge mit der maximalen Übereinstimmungslänge L. Diese ergibt sich für die Position $P = 7$ und die Länge $L = 5$ zu **Missi**.
- Dieser Schritt wird durch das *Triple* (7, 5, **s**) ausgedrückt $\Rightarrow$ allgemein (P, L, Z), wobei Z = **s** das Zeichen angibt, das nicht mehr mit der gefundenen Zeichenfolge im Suchpuffer übereinstimmt.
- Anschließend wird das Fenster um $L + 1 = 6$ Zeichen nach rechts verschoben. Im Vorschaupuffer steht nun **sippi–Mi**, im Suchpuffer **n–Missis** und die Codierung ergibt das Triple (2, 2, **p**).

Auf der nächsten Seite werden die LZ77–Codier & Decodier–Algorithmen genauer beschrieben.

LZ77 – die Grundform der Lempel–Ziv–Algorithmen (2)

Als weiteres Beispiel betrachten wir die LZ77–Codierung des Strings **ABABCBCBAABCABe** entsprechend der folgenden Grafik. Die Eingangsfolge hat die Länge $N = 15$. Weiter wird vorausgesetzt:

- Zeichen $Z \in \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{e}\}$, **e** entspricht *end–of–file* (Ende des Eingabe–Strings),
- Größe von Vorschau– und Suchpuffer jeweils $G = 4 \Rightarrow$ Position $P \in \{0, 1, 2, 3\}$.



Hierzu einige Anmerkungen (*Hinweis*: Der Decodiervorgang läuft in vergleichbarer Weise ab):

- Schritt 1 und 2: Es werden die Zeichen **A** und **B** durch die Triple (0, 0, **A**) und (0, 0, **B**) codiert, da diese im Suchpuffer noch nicht abgelegt sind. Dann Verschiebung des *Sliding Window* um 1.
- Schritt 3: **AB** wird über den Suchpuffer maskiert und gleichzeitig das noch unbekannte Zeichen **C** angehängt. Danach wird das *Sliding Window* um 3 Positionen nach rechts verschoben.
- Schritt 4: Hier wird gezeigt, dass der Suchstring **BCB** auch im Vorschaupuffer enden darf. Jetzt kann das Fenster um 4 Positionen verschoben werden.
- Schritt 5: Es wird im Suchpuffer lediglich **A** gefunden und **B** abgehängt. Bei größerem Suchpuffer könnten dagegen **ABC** gemeinsam maskiert werden. Dazu müsste $G \geq 7$ sein.
- Schritt 6: Ebenso muss das Zeichen **C** aufgrund des zu kleinen Puffers separat codiert werden. Da aber **CA** vorher noch nicht aufgetreten ist, würde $G = 7$ die Komprimierung nicht verbessern.
- Schritt 7: Mit der Berücksichtigung des *end–of–file* (**e**) gemeinsam mit **AB** aus dem Suchpuffer ist der Codiervorgang abgeschlossen.

Vor der Übertragung müssen natürlich die angegebenen Triple noch binär codiert werden. Dabei benötigt man im vorliegenden Beispiel für

- die Position $P \in \{0, 1, 2, 3\}$ zwei Bit (gelbe Hinterlegung),
- die Kopierlänge L drei Bit (grün hinterlegt), so dass man auch $L = 7$ noch darstellen könnte,
- alle Zeichen mit jeweils zwei Bit (weiß hinterlegt), z.B. **A** → **00**, **B** → **01**, **C** = **10**, **e** = **11**.

Damit hat die LZ77–Ausgangsfolge eine Länge von $7 \cdot 7 = 49$ Bit, während die Eingangsfolge nur $15 \cdot 2 = 30$ Bit benötigt $\Rightarrow$ **Eine LZ–Komprimierung macht nur bei großen Dateien Sinn.**

Die Lempel–Ziv–Variante LZ78 (1)

Der LZ77–Algorithmus erzeugt dann eine sehr ineffiziente Ausgabe, wenn sich häufigere Zeichenfolgen erst mit größerem Abstand wiederholen. Aufgrund der begrenzten Puffergröße G des *Sliding Window* können solche Wiederholungen oft nicht erkannt werden.

Lempel und Ziv haben dieses Manko bereits ein Jahr nach der Veröffentlichung der ersten Version LZ77 korrigiert. Der Algorithmus LZ78 verwendet zur Komprimierung anstelle des lokalen Wörterbuchs (Suchpuffer) ein globales Wörterbuch. Bei entsprechender Wörterbuchgröße lassen sich somit auch solche Phrasen, die schon längere Zeit vorher im Text aufgetaucht sind, effizient komprimieren.

Index I		Inhalt	Eintrag	LZ78–Coderausgabe	
dezimal	binär		im Schritt	formal	binär
0	0000	ε (leer)	$i = 0$	—	—
1	0001	A	$i = 1$	(0, A)	000000
2	0010	B	$i = 2$	(0, B)	000001
3	0011	AB	$i = 3$	(1, B)	000101
4	0100	C	$i = 4$	(0, C)	000010
5	0101	BC	$i = 5$	(2, C)	001010
6	0110	BA	$i = 6$	(2, A)	001000
7	0111	ABC	$i = 7$	(3, C)	001110
8	1000	ABe	$i = 8$	(3, e)	001111
9	1001	. . . .			

Wörterbuch

Zur Erklärung des LZ78–Algorithmus betrachten wir die gleiche Folge **ABABCBCBAABCABe** wie für das LZ77–Beispiel auf der letzten Seite.

- Die Grafik zeigt (mit roter Hinterlegung) das Wörterbuch mit Index I (in Dezimal– und Binärdarstellung, Spalte 1 und 2) und dem entsprechenden Inhalt (Spalte 3), der zum Codierschritt i eingetragen wird (Spalte 4). Bei LZ78 gilt sowohl für die Codierung als auch für die Decodierung stets $i = I$.
- In Spalte 5 findet man die formalisierte Coderausgabe (Index I, neues Zeichen Z). In der Spalte 6 ist die dazugehörige Binärcodierung angegeben mit vier Bit für den Index und der gleichen Zeichenzuordnung **A** → **00**, **B** → **01**, **C** → **10**, **e** („end–of–file") → **11** wie im letzten Beispiel.

Die Bildbeschreibung folgt auf der nächsten Seite.

Die Lempel–Ziv–Variante LZ78 (2)

Der LZ78–Algorithmus wird nun anhand dieses Beispiels wie folgt erklärt:

- Zu Beginn (Schritt $i = 0$) ist das Wörterbuch (WB) leer bis auf den Eintrag **ε** (leeres Zeichen, nicht zu verwechseln mit dem Leerzeichen, das aber hier nicht verwendet wird) mit Index $I = 0$.
- Im Schritt $i = 1$ findet man im Wörterbuch noch keinen verwertbaren Eintrag, und es wird (0, **A**) ausgegeben (**A** folgt auf **ε**). Im Wörterbuch erfolgt der Eintrag **A** in Zeile $I = 1$ (abgekürzt 1:**A**).
- Damit vergleichbar ist die Vorgehensweise im zweiten Schritt ($i = 2$). Ausgegeben wird hier (0, **B**) und ins Wörterbuch wird 2:**B** eingetragen.
- Da bei Schritt 3 bereits der Eintrag 1:**A** gefunden wird, können hier die Zeichen **AB** gemeinsam durch (1, **B**) codiert werden und es wird der neue Wörterbucheintrag 3:**AB** vorgenommen.
- Nach Codierung und Eintrag des neuen Zeichens **C** in Schritt 4 wird im Schritt 5 das Zeichenpaar **BC** gemeinsam codiert ⇒ (2, **C**) und in das Wörterbuch 5:**BC** eingetragen.
- In Schritt 6 werden mit **BA** ebenfalls zwei Zeichen gemeinsam behandelt und in den beiden letzten Schritten jeweils drei, nämlich 7:**ABC** und 8:**ABe**. Die Ausgabe (3, **C**) steht für „WB(3) + **C**" = **ABC** und die Ausgabe (3, **e**) für **ABe**.

Index I		Inhalt	Eintrag	LZ78–Coderausgabe	
dezimal	binär		im Schritt	formal	binär
0	0000	ε (leer)	$i = 0$	—	—
1	0001	A	$i = 1$	(0, A)	000000
2	0010	B	$i = 2$	(0, B)	000001
3	0011	AB	$i = 3$	(1, B)	000101
4	0100	C	$i = 4$	(0, C)	000010
5	0101	BC	$i = 5$	(2, C)	001010
6	0110	BA	$i = 6$	(2, A)	001000
7	0111	ABC	$i = 7$	(3, C)	001110
8	1000	ABe	$i = 8$	(3, e)	001111
9	1001	. . . .			

Wörterbuch

Im Beispiel besteht somit die LZ78–Codesymbolfolge aus $8 \cdot 6 = 48$ Bit. Das Ergebnis ist vergleichbar mit LZ77 (49 Bit). Auf Details und Verbesserungen von LZ78 wird hier verzichtet. Hier verweisen wir auf den **LZW–Algorithmus,** der auf den nächsten Seiten beschrieben wird. Soviel nur vorneweg:

- Der Index I wird hier einheitlich mit 4 Bit dargestellt, wodurch das Wörterbuch auf 16 Einträge beschränkt ist. Durch eine **variable Bitanzahl** für den Index kann man diese Einschränkung umgehen. Gleichzeitig erhält man so einen besseren Komprimierungsfaktor.
- Das Wörterbuch muss bei allen LZ–Varianten nicht übertragen werden, sondern wird beim Decoder in genau gleicher Weise erzeugt wie auf der Coderseite. Die Decodierung erfolgt bei LZ78 – nicht aber bei LZW – ebenfalls in analoger Weise wie die Codierung.
- Alle LZ–Verfahren sind asymptotisch optimal, das heißt, dass bei unendlich langen Folgen die mittlere Codewortlänge pro Quellensymbol gleich der Quellenentropie ist: $L_M = H$. Bei kurzen Folgen ist die Abweichung allerdings beträchtlich. Mehr dazu am **Kapitelende.**

Der Lempel–Ziv–Welch–Algorithmus (1)

Die heute gebräuchlichste Variante der Lempel–Ziv–Komprimierung wurde von **Terry Welch** entworfen und 1983 veröffentlicht. Wir nennen diese den *Lempel–Ziv–Welch–Algorithmus*, abgekürzt mit LZW. Ebenso wie LZ78 leichte Vorteile gegenüber LZ77 aufweist (wie zu erwarten – warum sonst hätte der Algorithmus modifiziert werden sollen?), hat LZW gegenüber LZ78 auch mehr Vorteile als Nachteile.

Schritt Nr.	String	LZW–Coderausgabe			Wörterbuch			
		Index I	binär	verkürzt	Index I	binär	verkürzt	Inhalt
$i = 0$:	Vorbelegung des Wörterbuchs				$I = 0$	0000	0	A
					$I = 1$	0001	1	B
					$I = 2$	0010	10	C
					$I = 3$	0011	11	e
$i = 1$	A	$I = 0$	0000	00	$I = 4$	0100	100	AB
$i = 2$	B	$I = 1$	0001	001	$I = 5$	0101	101	BA
$i = 3$	AB	$I = 4$	0100	100	$I = 6$	0110	110	ABC
$i = 4$	C	$I = 2$	0010	010	$I = 7$	0111	111	CB
$i = 5$	B	$I = 1$	0001	001	$I = 8$	1000	1000	BC
$i = 6$	CB	$I = 7$	0111	0111	$I = 9$	1001	1001	CBA
$i = 7$	A	$I = 0$	0000	0000	$I = 10$	1010	1010	AA
$i = 8$	ABC	$I = 6$	0110	0110	$I = 11$	1011	1011	ABCA
$i = 9$	AB	$I = 4$	0100	0100	$I = 12$	1100	1100	ABe
$i = 10$	e	$I = 3$	0011	0011	wg. „end–of–file“			

Die Grafik zeigt die Coderausgabe für die beispielhafte Eingangsfolge **ABABCBCBAABCABe**. Rechts dargestellt ist das Wörterbuch (rot hinterlegt), das bei der LZW–Codierung sukzessive entsteht. Die Unterschiede gegenüber LZ78 erkennt man im Vergleich zur **Grafik auf der letzten Seite,** nämlich:

- Bei LZW sind im Wörterbuch schon zu Beginn ($i = 0$) alle vorkommenden Zeichen eingetragen und einer Binärfolge zugeordnet, im Beispiel mit den Indizes $I = 0, \ldots, I = 3$.
- Das bedeutet aber auch, dass bei LZW doch gewisse Kenntnisse über die Nachrichtenquelle vorhanden sein müssen, während LZ78 eine „echte universelle Codierung” darstellt.
- Bei LZW wird zu jedem Codierschritt i nur ein Wörterbuchindex I übertragen, während bei LZ78 die Kombination (I, Z) ausgegeben wird; Z gibt dabei das aktuell neue Zeichen an.
- Aufgrund des Fehlens von Z in der Coderausgabe ist die LZW–Decodierung komplizierter als bei LZ78. Nähere Angaben zur LZW–Decodierung finden Sie auf **Seite 6** des Kapitels.

Der Lempel–Ziv–Welch–Algorithmus (2)

Für die nachfolgende beispielhafte LZW–Codierung wird wie bei der Beschreibung von LZ77 und LZ78 wieder die Eingangsfolge **ABABCBCBAABCABe** vorausgesetzt.

Schritt Nr.	String	LZW–Coderausgabe			Wörterbuch			
		Index I	binär	verkürzt	Index I	binär	verkürzt	Inhalt
$i = 0$:	Vorbelegung des Wörterbuchs				$I = 0$	0000	0	A
					$I = 1$	0001	1	B
					$I = 2$	0010	10	C
					$I = 3$	0011	11	e
$i = 1$	A	$I = 0$	0000	00	$I = 4$	0100	100	AB
$i = 2$	B	$I = 1$	0001	001	$I = 5$	0101	101	BA
$i = 3$	AB	$I = 4$	0100	100	$I = 6$	0110	110	ABC
$i = 4$	C	$I = 2$	0010	010	$I = 7$	0111	111	CB
$i = 5$	B	$I = 1$	0001	001	$I = 8$	1000	1000	BC
$i = 6$	CB	$I = 7$	0111	0111	$I = 9$	1001	1001	CBA
$i = 7$	A	$I = 0$	0000	0000	$I = 10$	1010	1010	AA
$i = 8$	ABC	$I = 6$	0110	0110	$I = 11$	1011	1011	ABCA
$i = 9$	AB	$I = 4$	0100	0100	$I = 12$	1100	1100	ABe
$i = 10$	e	$I = 3$	0011	0011	wg. „end–of–file“			

- Schritt $i = 0$ (Vorbelegung): Die erlaubten Zeichen **A**, **B**, **C** und **e** („end–of–file”) werden in das Wörterbuch eingetragen und den Indizes $I = 0$, ... , $I = 3$ zugeordnet.
- Schritt $i = 1$: **A** wird durch den Dezimalindex $I = 0$ codiert und dessen Binärdarstellung **0000** übertragen. Anschließend wird ins Wörterbuch die Kombination aus dem aktuellen Zeichen **A** und dem nachfolgenden Zeichen **B** der Eingangsfolge unter dem Index $I = 4$ abgelegt.
- Schritt $i = 2$: Darstellung von **B** durch Index $I = 1$ bzw. **0001** (binär) sowie Wörterbucheintrag von **BA** mit Index $I = 5$.
- Schritt $i = 3$: Aufgrund des Wörterbucheintrags **AB** zum Zeitpunkt $i = 1$ ergibt sich der zu übertragende Index $I = 4$ (binär: **0100**). Danach wird ins Wörterbuch **ABC** neu eingetragen.
- Schritt $i = 8$: Hier werden die Zeichen **ABC** gemeinsam durch den Index $I = 6$ (binär: **0110**) dargestellt und der Eintrag für **ABCA** vorgenommen.

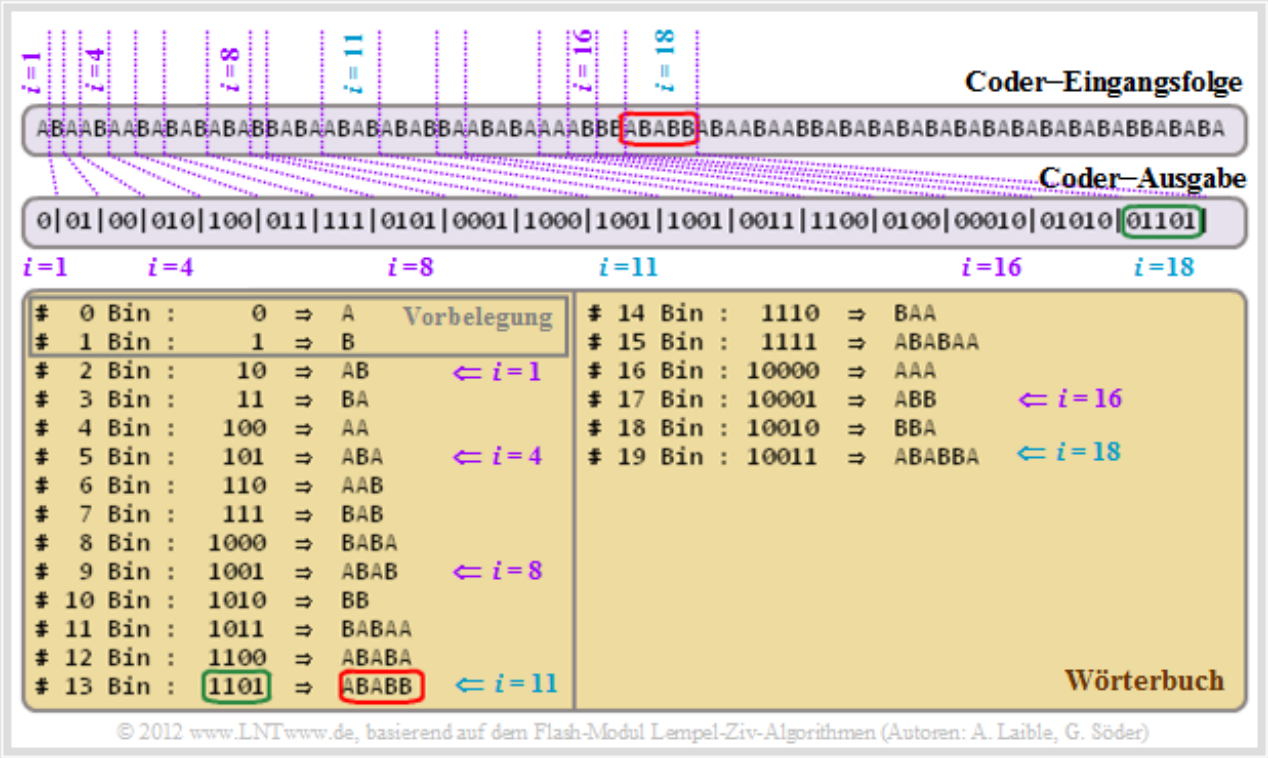
Mit der Codierung von **e** (EOF–Marke) ist der Codiervorgang nach 10 Schritten beendet. Bei LZ78 wurden nur 8 Schritte benötigt. Es ist aber zu berücksichtigen:

- Der LZW–Algorithmus benötigt für die Darstellung dieser 15 Eingangssymbole nur $10 \cdot 4 = 40$ Bit gegenüber den $8 \cdot 6 = 48$ Bit bei LZ78. Vorausgesetzt ist für diese einfache Rechnung jeweils 4 Bit zur Indexdarstellung.
- Sowohl bei LZW als auch bei LZ78 kommt man mit weniger Bit aus (nämlich mit 34 bzw. 42), wenn man berücksichtigt, dass zum Schritt $i = 1$ der Index nur mit 2 Bit codiert werden muss ($I \le 3$) und für $i = 2$ bis $i = 5$ auch 3 Bit ausreichen ($I \le 7$).

Auf den beiden folgenden Seiten wird auf die variable Bitanzahl zur Indexdarstellung sowie auf die Decodierung von LZ78– und LZW–codierten Binärfolgen noch im Detail eingegangen.

Lempel–Ziv–Codierung mit variabler Indexbitlänge

Aus Gründen einer möglichst kompakten Darstellung betrachten wir nun nur noch Binärquellen mit dem Wertevorrat {**A**, **B**}. Auch das Abschlusszeichen **end–of–file** bleibt unberücksichtigt.



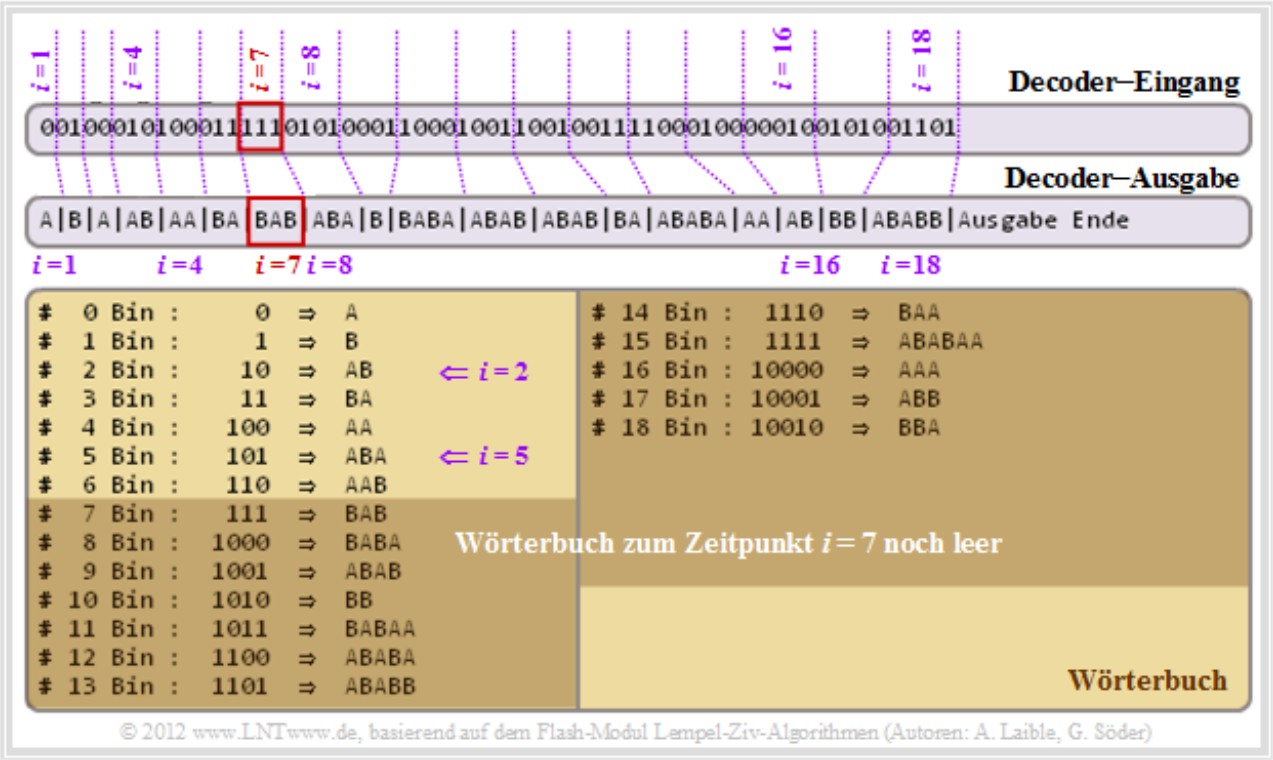
Wir betrachten die LZW–Codierung anhand eines Bildschirmabzugs unseres interaktiven Flash–Moduls **Lempel–Ziv–Algorithmen.** Die Aussagen gelten aber in gleicher Weise für LZ78.

- Beim ersten Codierschritt ($i = 1$) wird **A** mit **0** codiert. Danach erfolgt im Wörterbuch der Eintrag mit dem Index $I = 2$ und dem Inhalt **AB**.
- Da es bei Schritt $i = 1$ im Wörterbuch mit **A** und **B** nur zwei Einträge gibt, genügt ein Bit. Dagegen werden bei Schritt 2 und 3 für **B** ⇒ **01** bzw. **A** ⇒ **00** jeweils zwei Bit benötigt.
- Ab $i = 4$ erfolgt die Indexdarstellung mit 3 Bit, ab $i = 8$ mit 4 Bit und ab $i = 16$ mit 5 Bit. Hieraus lässt sich ein einfacher Algorithmus für die jeweilige Index–Bitanzahl $L(i)$ ableiten.
- Betrachten wir abschließend den Codierschritt $i = 18$. Hier wird die rot markierte Sequenz **ABABB**, die zum Zeitpunkt $i = 11$ in das Wörterbuch eingetragen wurde (Index $I = 13$ ⇒ **1101**) bearbeitet. Die Ausgabe lautet wegen $i \geq 16$ aber nun **01101** (grüne Markierung).

Die Verbesserung durch variable Indexbitlänge ist auch bei LZ78 in gleicher Weise möglich.

Decodierung des LZW–Algorithmus

Am Decoder liegt nun die auf der **letzten Seite** ermittelte Coder–Ausgabe als Eingangsfolge an. Die Grafik zeigt, dass es auch bei variabler Indexbitlänge möglich ist, diese Folge eindeutig zu decodieren.



Beim Decoder wird genau das gleiche Wörterbuch generiert wie beim Coder, doch erfolgen hier die Wörterbucheinträge einen Zeitschritt später. Weiter gilt:

- Dem Decoder ist bekannt, dass im ersten Codierschritt der Index I mit nur einem Bit codiert wurde, in den Schritten 2 und 3 mit zwei Bit, ab $i = 4$ mit drei Bit, ab $i = 8$ mit vier Bit, usw.
- Zum Schritt $i = 1$ wird also **0** als **A** decodiert. Ebenso ergibt sich zum Schritt $i = 2$ aus der Vorbelegung des Wörterbuches und der vereinbarten Zwei–Bit–Darstellung: **1 ⇒ 01 ⇒ B**.
- Der Eintrag der Zeile $I = 2$ (Inhalt: **AB**) des Wörterbuchs erfolgt also erst zum Schritt $i = 2$, während beim **Codiervorgang** dies bereits am Ende von Schritt 1 geschehen konnte.
- Betrachten wir weiter die Decodierung für $i = 4$. Der Index 2 liefert das Decodierergebnis **AB** und im nächsten Schritt ($i = 5$) wird die Wörterbuchzeile $I = 5$ mit **ABA** belegt.
- Diese Zeitverschiebung hinsichtlich der WB–Einträge kann zu Decodierproblemen führen. Zum Beispiel gibt es zum Schritt $i = 7$ noch keinen Wörterbuch–Eintrag mit Index $I = 7$.
- Was ist in einem solchen Fall ($I = i$) zu tun? Man nimmt in diesem Fall das Ergebnis des vorherigen Decodierschrittes (hier: **BA** für $i = 6$) und fügt das erste Zeichen dieser Sequenz am Ende noch einmal an. Man erhält so das Decodierergebnis für $i = 7$ zu **BAB**.
- Natürlich ist es unbefriedigend, nur ein Rezept anzugeben. In der **Aufgabe Z2.4** sollen Sie das Vorgehen selbst begründen. Wir verweisen hier auf die **Musterlösung** zur Aufgabe.

Bei der LZ78–Decodierung tritt das hier geschilderte Problem nicht auf, da nicht nur der Index I, sondern auch das aktuelle Zeichen Z im Codierergebnis enthalten ist und übertragen wird.

Effizienz der Lempel–Ziv–Codierung (1)

Für den Rest dieses Kapitels gehen wir von folgenden Voraussetzungen aus:

- Der *Symbolumfang* der Quelle (oder im übertragungstechnischen Sinne die Stufenzahl) sei M, wobei M eine Zweierpotenz darstellt $\Rightarrow$ $M = 2, 4, 8, 16, ...$
- Die Quellenentropie sei H. Gibt es keine statistischen Bindungen zwischen den Symbolen, so gilt $H = H_0$, wobei $H_0 = \text{ld}\ M$ den *Entscheidungsgehalt* angibt. Andernfalls gilt $H < H_0$.
- Eine Symbolfolge der Länge N wird quellencodiert und liefert eine binäre Codefolge der Länge L. Über die Art der Quellencodierung treffen wir vorerst keine Aussage.

Nach dem **Quellencodierungstheorem** muss die mittlere Codewortlänge L_M größer oder gleich der Quellenentropie H (in bit/Quellensymbol) sein. Das bedeutet

- für die *Gesamtlänge* der quellencodierten Binärfolge:

$$L \ge N \cdot H\,,$$

- für die *relative Redundanz* der Codefolge, im Folgenden kurz **Restredundanz** genannt:

$$r = \frac{L - N \cdot H}{L}\,.$$

Beispiel: Gäbe es für eine redundanzfreie binäre Quellensymbolfolge ($M = 2, p_A = p_B = 0.5$, ohne statistische Bindungen) der Länge $N = 10000$ eine *perfekte Quellencodierung*, so hätte auch die Codefolge die Länge $L = 10000$.

- Für diese Nachrichtenquelle ist Lempel–Ziv nicht geeignet. Es wird $L > N$ gelten. Man kann es auch ganz lapidar ausdrücken: Die perfekte Quellencodierung ist hier gar keine Codierung.
- Eine redundante Binärquelle mit $p_A = 0.89, p_B = 0.11 \Rightarrow H = 0.5$ könnte man mit einer perfekten Quellencodierung durch $L = 5000$ Bit darstellen, ohne dass wir hier sagen können, wie diese perfekte Quellencodierung aussieht.
- Bei einer Quaternärquelle ist $H > 1$ (bit/Quellensymbol) möglich, so dass auch bei perfekter Codierung stets $L > N$ sein wird. Ist die Quelle redundanzfrei (keine Bindungen, alle M Symbole gleichwahrscheinlich), so hat sie die Entropie $H = 2$ bit/Quellensymbol.

Bei allen diesen Beispielen für perfekte Quellencodierung wäre die relative Redundanz der Codefolge (Restredundanz) $r = 0$. Das heißt: Die Nullen und Einsen sind gleichwahrscheinlich und es bestehen keine statistischen Bindungen zwischen einzelnen Symbolen.

Das Problem ist: Bei endlicher Folgenlänge N gibt es keine perfekte Quellencodierung.

Effizienz der Lempel–Ziv–Codierung (2)

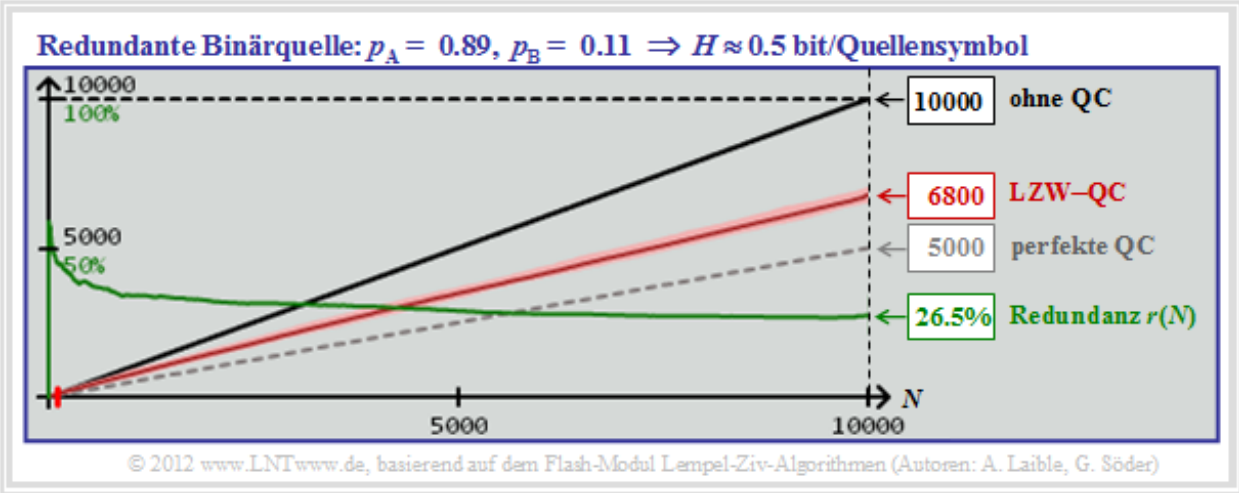
Von den Lempel–Ziv–Algorithmen weiß man (und kann diese Aussage sogar beweisen), dass sie **asymptotisch optimal** sind. Das bedeutet, dass die relative Redundanz der Codesymbolfolge

$$r(N) = \frac{L(N) - N \cdot H}{L(N)} = 1 - \frac{N \cdot H}{L(N)}$$

(hier als Funktion der Quellensymbolfolgenlänge N geschrieben) für große N den Grenzwert 0 liefert:

$$\lim_{N \to \infty} r(N) = 0.$$

Was aber sagt die Eigenschaft „asymptotisch optimal" für praxisrelevante Folgenlängen aus? Nicht allzu viel, wie der nachfolgende Bildschirmabzug des Flash–Moduls **Lempel–Ziv–Algorithmen** zeigt. Die Kurven gelten für den **LZW–Algorithmus.** Die Ergebnisse für **LZ77** und **LZ78** sind aber nur geringfügig schlechter.



Diese Grafik (und auch die Grafiken auf den nächsten Seiten) zeigen die Abhängigkeit der folgenden Größen von der Quellensymbolfolgenlänge N:

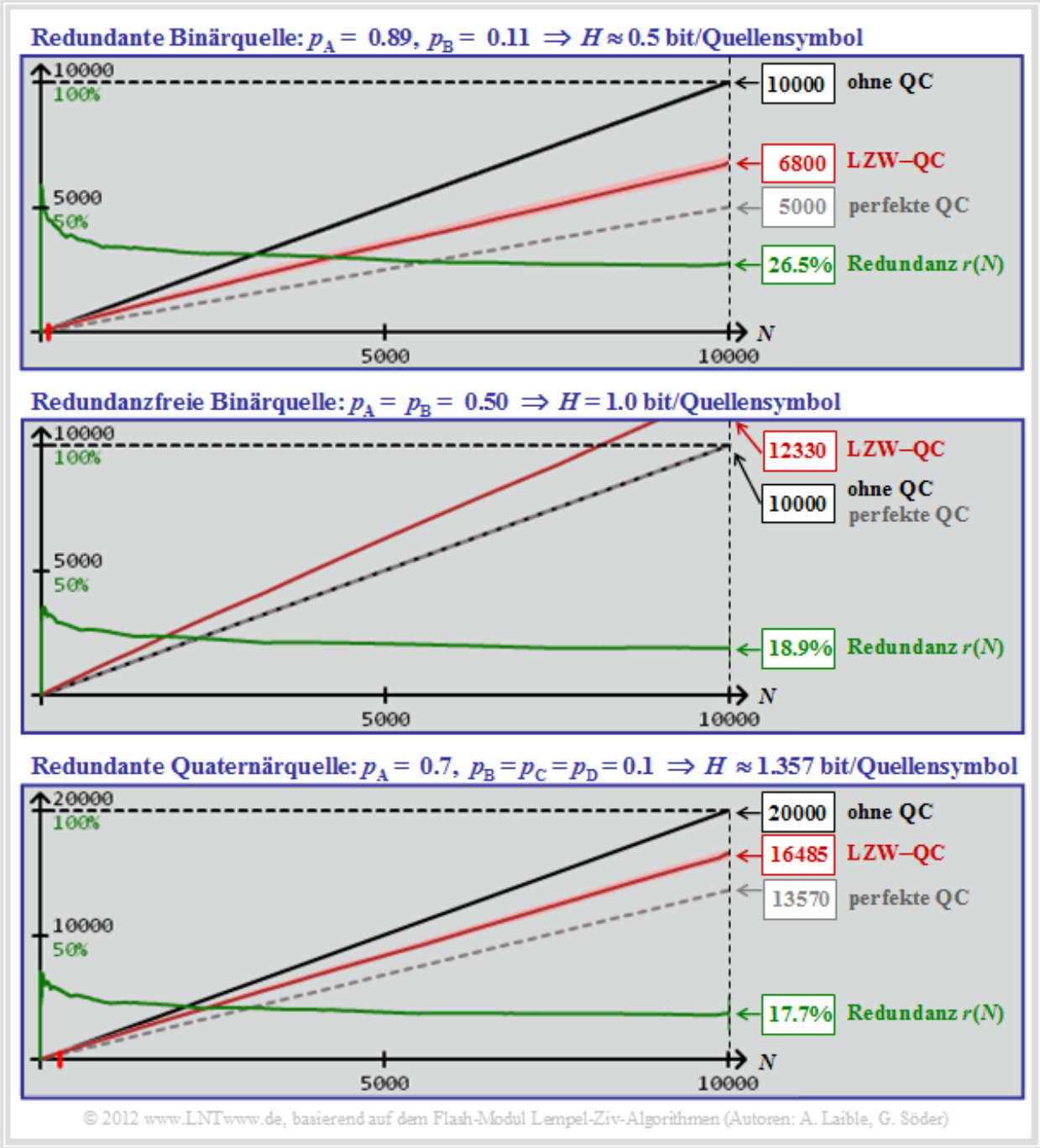
- die erforderliche Bitanzahl ($N \cdot$ ld M) ohne Quellencodierung (schwarze Kurven),
- die erforderliche Bitanzahl ($H \cdot N$) bei perfekter Quellencodierung (grau–gestrichelt),
- die erforderliche Bitanzahl $L(N)$ bei LZW–Codierung (rote Kurven nach Mittelung),
- die relative Redundanz $r(N)$ bei LZW–Codierung (grüne Kurven).

Die Grafik auf dieser Seite gilt für eine redundante Binärquelle ($M = 2$) mit der Quellenentropie $H = 0.5$. Man erkennt:

- Die schwarze und die graue Kurve sind echte Gerade (nicht nur bei diesem Parametersatz).
- Die rote Kurve $L(N)$ zeigt eine leichte Krümmung (mit bloßem Auge schwer zu erkennen).
- Wegen dieser Krümmung von $L(N)$ fällt die grüne Kurve $r(N) = 1 – 0.5 \cdot N/L(N)$ leicht ab.
- Abzulesen sind die Zahlenwerte $L(N = 10000) = 6800$ und $r(N = 10000) = 26.5\%$.

Effizienz der Lempel–Ziv–Codierung (3)

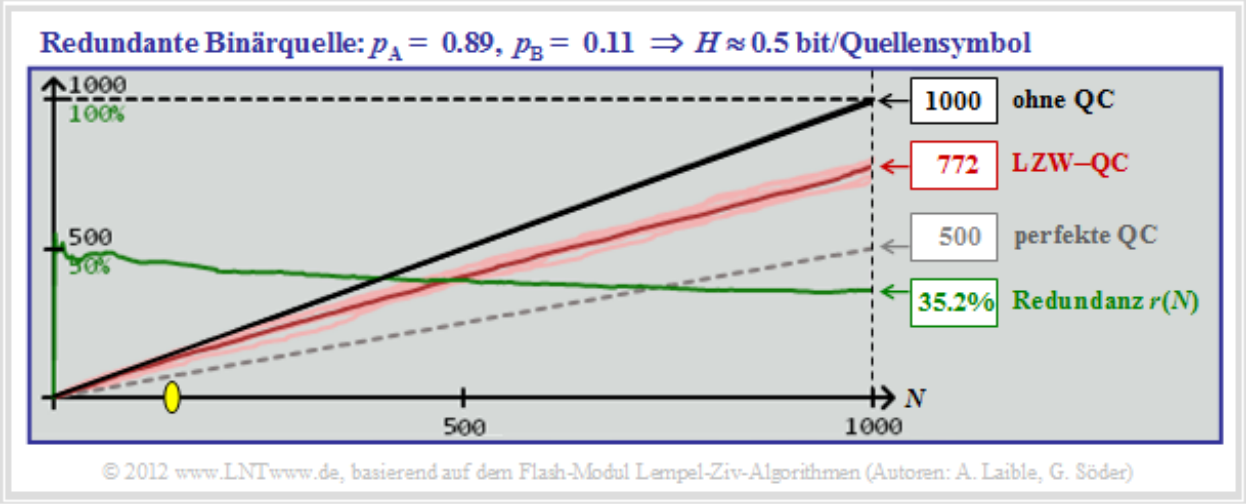
In der oberen Grafik ist nochmals die redundante Binärquelle mit $H = 0.5$ dargestellt. Die mittlere Grafik gilt dagegen für gleichwahrscheinliche Binärsymbole $\Rightarrow H = 1$. Hier fallen die graue und die schwarze Gerade zusammen und die leicht gekrümmte rote Kurve liegt erwartungsgemäß darüber. Obwohl hier die LZW–Codierung eine Verschlechterung bringt – erkennbar aus der Angabe $L(N = 10000) = 12330$, ist die relative Redundanz mit $r(N = 10000) = 18.9\%$ kleiner als bei der oberen Grafik.



Bei einer redundanten Quaternärquelle mit $H = 1.357$ wären entsprechend der unteren Grafik ohne Codierung 20000 Bit (für $N = 10000$) erforderlich und mit LZW–Codierung nur $L \approx 16485$. Die relative Redundanz beträgt hier $r(N = 10000) = 17.7\%$.

Quantitative Aussagen zur asymptotischen Optimalität

Die Ergebnisse der letzten Seite haben gezeigt, dass die relative Restredundanz $r(N = 10000)$ deutlich größer ist als der theoretisch versprochene Wert $r(N \to \infty) = 0$. Dieses praxisrelevante Ergebnis soll nun am Beispiel der redundanten Binärquelle mit $H = 0.5$ bit/Quellensymbol präzisiert werden.



Die Grafik zeigt jeweils Simulationen mit $N = 1000$ Binärsymbolen, wobei sich nach Mittelung über 10 Versuchsreihen $r(N = 1000) = 35.2\%$ ergibt. Unterhalb des gelben Punktes (im Beispiel bei $N \approx 150$) bringt der LZW–Algorithmus sogar eine Verschlechterung. In diesem Bereich gilt nämlich $L > N$.

Die Tabelle fasst die Simulationsergebnisse für die redundante Binärquelle ($H = 0.5$) zusammen:

- In der Zeile 4 ist die Restredundanz $r(N)$ für verschiedene Folgenlängen N zwischen 1000 und 50000 angegeben. Man erkennt den nur langsamen Abfall mit steigendem N.
- Entsprechend Literaturangaben nimmt die Restredundanz mit $1/\lg(N)$ ab. In Zeile 5 sind die Ergebnisse einer empirischen Formel eingetragen (Anpassung für $N = 10000$):

$$r'(N) = A/\lg\,(N) \quad \text{mit} \quad A = r(N = 10000) \cdot \lg\,10000 = 0.265 \cdot 4 = 1.06\,.$$

N: Eingangsfolgenlänge	1000	2000	5000	10000	20000	50000	10^6	10^9	10^{12}
$L(N)$: Ausgangsfolgenlänge	772	1488	3516	6800	13263	32100			
$K(N)$: Komprimierungsfaktor	0.772	0.680	0.680	0.680	0.680	0.680			
$r(N)$: relative Restredundanz	0.352	0.328	0.289	0.265	0.242	0.221			
$r'(N)$: empirische Faustformel	0.353	0.321	0.287	0.265	0.246	0.226	0.177	0.118	0.088

Anpassung

Man erkennt die gute Übereinstimmung zwischen unseren Simulationsergebnissen $r(N)$, basierend auf unserem Interaktionsmodul **Lempel–Ziv–Algorithmen,** und der Faustformel $r'(N)$. Man erkennt aber auch, dass für $N = 10^{12}$ die Restredundanz des LZW–Algorithmus noch immer 8.8% beträgt.

Bei anderen Quellen erhält man mit anderen Zahlenwerten des Parameters A ähnliche Ergebnisse. Der prinzipielle Kurvenverlauf bleibt aber gleich. Siehe auch **Aufgabe A2.5** und **Aufgabe Z2.5.**

Der Huffman–Algorithmus

Wir setzen nun voraus, dass die Quellensymbole q_ν einem Alphabet $\{q_\mu\} = \{\mathbf{A}, \mathbf{B}, \mathbf{C}, ...\}$ mit dem Symbolumfang M entstammen und statistisch voneinander unabhängig seien.

Beispielsweise gelte für den Symbolumfang $M = 8$:

$$\{q_\mu\} = \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}, \mathbf{G}, \mathbf{H}\}.$$

David A. Huffman hat 1952 – also kurz nach Shannons bahnbrechenden Veröffentlichungen – einen Algorithmus zur Konstruktion von optimalen präfixfreien Codes angegeben.

Dieser *Huffman–Algorithmus* soll hier ohne Herleitung und Beweis angegeben werden, wobei wir uns hier auf Binärcodes beschränken. Das heißt: Für die Codesymbole gelte stets $c_\nu \in \{\mathbf{0}, \mathbf{1}\}$. Hier ist das Rezept:

- Man ordne die Symbole nach fallenden Auftrittswahrscheinlichkeiten.
- Man fasse die zwei unwahrscheinlichsten Symbole zu einem neuen Symbol zusammen.
- Man wiederhole (1) und (2), bis nur mehr zwei (zusammengefasste) Symbole übrig bleiben.
- Man codiert die wahrscheinlichere Symbolmenge mit **1** und die andere Menge mit **0**.
- Man ergänzt in Gegenrichtung (also von unten nach oben) die jeweiligen Binärcodes der aufgespaltenen Teilmengen entsprechend den Wahrscheinlichkeiten mit **1** bzw. **0**.

Beispiel: Ohne Einschränkung der Allgemeingültigkeit setzen wir voraus, dass die $M = 6$ Symbole **A**, ... , **F** bereits entsprechend ihren Wahrscheinlichkeiten geordnet sind:

$$p_{\rm A} = 0.30, \quad p_{\rm B} = 0.24, \quad p_{\rm C} = 0.20, \quad p_{\rm D} = 0.12, \quad p_{\rm E} = 0.10, \quad p_{\rm F} = 0.04.$$

Durch paarweises Zusammenfassen und anschießendem Sortieren erhält man in fünf Schritten die folgenden Symbolkombinationen (resultierende Wahrscheinlichkeiten in Klammern):

1. **A** (0.30), **B** (0.24), **C** (0.20), **EF** (0.14), **D** (0.12),
2. **A** (0.30), **EFD** (0.26), **B** (0.24), **C** (0.20),
3. **BC** (0.44), **A** (0.30), **EFD** (0.26),
4. **AEFD** (0.56), **BC** (0.44),
5. Root **AEFDBC** (1.00).

Rückwärts (gemäß den Schritten 5 bis 1) erfolgt dann die Zuordnung zu Binärsymbolen. Ein „*x*" weist darauf hin, dass in den nächsten Schritten noch Bits hinzugefügt werden müssen:

5. **AEFD** → **1***x*, **BC** → **0***x*,
4. **A** → **11**, **EFD** → **10***x*,
3. **B** → **01**, **C** → **00**,
2. **EF** → **101***x*, **D** → **100**,
1. **E** → **1011**, **F** → **1010**.

Die Unterstreichungen markieren die endgültige Binärcodierung.

Zum Begriff „Entropiecodierung"

Wir gehen weiterhin von den Wahrscheinlichkeiten und Zuordnungen des letzten Beispiels aus:

$$p_{\rm A} = 0.30\,,\ p_{\rm B} = 0.24\,,\ p_{\rm C} = 0.20\,,\ p_{\rm D} = 0.12\,,\ p_{\rm E} = 0.10\,,\ p_{\rm F} = 0.04\,;$$
$$\mathbf{A} \rightarrow \mathbf{11}\,,\ \mathbf{B} \rightarrow \mathbf{01}\,,\ \mathbf{C} \rightarrow \mathbf{00}\,,\ \mathbf{D} \rightarrow \mathbf{100}\,,\ \mathbf{E} \rightarrow \mathbf{1011}\,,\ \mathbf{F} \rightarrow \mathbf{1010}\,.$$

Von den sechs Quellensymbolen werden also drei mit je zwei Bit, eines mit drei Bit und zwei Symbole (**E** und **F**) mit vier Bit codiert. Die mittlere Codewortlänge ergibt sich damit zu

$$L_{\rm M} = (0.30 + 0.24 + 0.20) \cdot 2 + 0.12 \cdot 3 + (0.10 + 0.04) \cdot 4 = 2.4\,{\rm bit/Quellensymbol}\,.$$

Aus dem Vergleich mit der Quellenentropie H = 2.365 bit/Quellensymbol erkennt man die Effizienz der Huffman–Codierung.

Merke: Es gibt keinen präfixfreien (⇒ sofort decodierbaren) Code, der allein unter Ausnutzung der Auftrittswahrscheinlichkeiten zu einer kleineren mittleren Codewortlänge führt als der Huffman–Code.

In diesem Sinne ist der Huffman–Code optimal. Wären die Symbolwahrscheinlichkeiten

$$p_{\rm A} = p_{\rm B} = p_{\rm C} = 1/4\,,\ p_{\rm D} = 1/8\,,\ p_{\rm E} = p_{\rm F} = 1/16\,,$$

so würde für die Entropie und für die mittlere Codewortlänge gleichermaßen gelten:

$$H = 3 \cdot 1/4 \cdot {\rm ld}\,(4) + 1/8 \cdot {\rm ld}\,(8) + 2 \cdot 1/16 \cdot {\rm ld}\,(16) = 2.375\,{\rm bit/Quellensymbol}\,,$$
$$L_{\rm M} = 3 \cdot 1/4 \cdot 2 + 1/8 \cdot 3 + 2 \cdot 1/16 \cdot 4 = 2.375\,{\rm bit/Quellensymbol}\,.$$

Hinweis: Aus Platzgründen ist hier der *Logarithmus dualis* „$\log_2$" mit „ld" bezeichnet.

Aus dieser Eigenschaft erklärt sich der Begriff **Entropiecodierung**. Man versucht bei dieser Form von Quellencodierung, die Länge L_μ der Binärfolge (bestehend aus Nullen und Einsen) für das Symbol q_μ gemäß der Entropieberechnung wie folgt an dessen Auftrittswahrscheinlichkeit p_μ anzupassen:

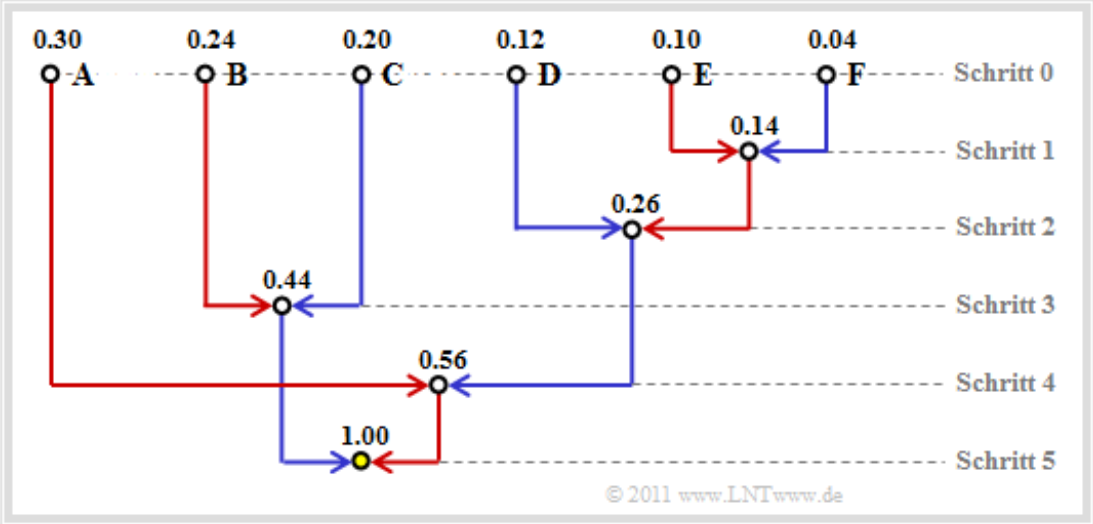
$$L_\mu = \log_2\,(1/p_\mu)\,.$$

Natürlich gelingt das nicht immer, sondern nur dann, wenn alle Auftrittswahrscheinlichkeiten p_μ in der Form 2^{-k} (k = 1, 2, 3, ...) dargestellt werden können. In diesem Sonderfall – und nur in diesem – stimmt die mittlere Codewortlänge $L_{\rm M}$ exakt mit der Quellenentropie H überein (siehe zweites Zahlenbeispiel).

Nach dem **Quellencodierungstheorem** gibt es keinen (decodierbaren) Code, der im Mittel mit weniger Binärzeichen pro Quellensymbol auskommt.

Darstellung des Huffman–Codes als Baumdiagramm (1)

Häufig wird für die Konstruktion des Huffman–Codes eine **Baumstruktur** verwendet.Für das bisher betrachtete **Beispiel** zeigt diese die folgende Grafik:



Man erkennt:

- Bei jedem Schritt des Huffman–Algorithmus werden die beiden Zweige mit den jeweils kleinsten Wahrscheinlichkeiten zusammengefasst. Der Knoten im Schritt 1 fasst die zwei Symbole **E** und **F** mit den aktuell kleinsten Wahrscheinlichkeiten zusammen. Dieser Knoten ist mit $p_{\rm E} + p_{\rm F} = 0.14$ beschriftet.
- Der vom Symbol mit der kleineren Wahrscheinlichkeit (hier **F**) zum Summenknoten verlaufende Zweig ist blau eingezeichnet, der andere rot.

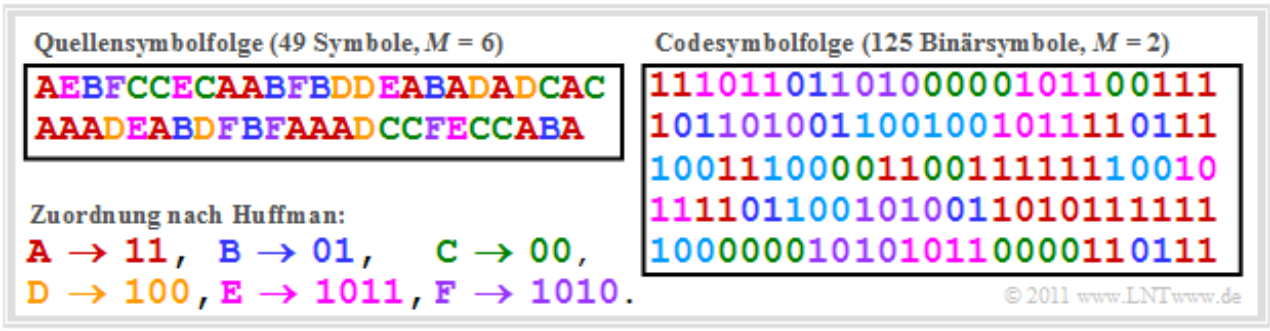
Nach fünf Schritten ist man bei der Baumwurzel („*Root*”) mit der Gesamtwahrscheinlichkeit 1 angelangt. Verfolgt man nun den Verlauf von der Wurzel (in obiger Grafik mit gelber Füllung) zu den einzelnen Symbolen zurück, so kann man aus den Farben der einzelnen Zweige die Symbolzuordnung ablesen. Mit den Zuordnungen „rot” → **1** und „blau” → **0** ergibt sich beispielsweise von der Wurzel zu Symbol

- **A**: rot, rot → **11**,
- **B**: blau, rot → **01**,
- **C**: blau, blau → **00**,
- **D**: rot, blau, blau → **100**,
- **E**: rot, blau, rot, rot → **1011**,
- **F**: rot, blau, rot, blau → **1010**.

Die Zuordnung „rot” → **0** und „blau” → **1** würde ebenfalls zu einem optimalen präfixfreien Huffman–Code führen.

Darstellung des Huffman–Codes als Baumdiagramm (2)

Die folgende Grafik zeigt die Huffman–Codierung von 49 Symbolen $q_\nu \in \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}\}$ mit der auf der letzten Seite hergeleiteten Zuordnung. Die binäre Codesymbolfolge weist die mittlere Codewortlänge $L_\text{M} = 125/49 = 2.551$ auf. Die Farben dienen ausschließlich zur besseren Orientierung.



Aufgrund der kurzen Quellensymbolfolge ($N = 49$) weichen die Auftrittshäufigkeiten h_A, ... , h_F der simulierten Folgen signifikant von den vorgegebenen Wahrscheinlichkeiten p_A, ... , p_F ab:

$$p_\text{A} = 0.30 \Rightarrow h_\text{A} = 16/49 \approx 0.326, \quad p_\text{B} = 0.24 \Rightarrow h_\text{B} = 7/49 \approx 0.143,$$
$$p_\text{C} = 0.24 \Rightarrow h_\text{C} = 9/49 \approx 0.184, \quad p_\text{D} = 0.12 \Rightarrow h_\text{D} = 7/49 \approx 0.143,$$
$$p_\text{E} = 0.10 \Rightarrow h_\text{E} = 5/49 \approx 0.102, \quad p_\text{F} = 0.04 \Rightarrow h_\text{E} = 5/49 \approx 0.102.$$

Damit ergibt sich ein etwas größerer Entropiewert:

$$H(\text{bezüglich } p_\mu) = 2.365 \text{ bit/Quellensymbol}$$
$$\Rightarrow \quad H(\text{bezüglich } h_\mu) = 2.451 \text{ bit/Quellensymbol}.$$

Würde man den Huffman–Code mit diesen „neuen” Wahrscheinlichkeiten h_A, ... , h_F bilden, so ergäben sich folgende Zuordnungen:

$$\mathbf{A} \rightarrow 11, \ \mathbf{B} \rightarrow 100, \ \mathbf{C} \rightarrow 00, \ \mathbf{D} \rightarrow 101, \ \mathbf{E} \rightarrow 010, \ \mathbf{F} \rightarrow 011.$$

Nun würden nur **A** und **C** mit zwei Bit dargestellt, die anderen vier Symbole durch jeweils drei Bit. Die Codesymbolfolge hätte dann eine Länge von $(16 + 9) \cdot 2 + (7 + 7 + 5 + 5) \cdot 3 = 122$ Bit, wäre also um drei Bit kürzer als nach der bisherigen Codierung. Die mittlere Codewortlänge wäre dann $L_\text{M} = 122/49 \approx 2.49$ bit/Quellensymbol anstelle von $L_\text{M} \approx 2.55$ bit/Quellensymbol.

Dieses Beispiel lässt sich wie folgt interpretieren:

- Die Huffman–Codierung lebt von der (genauen) Kenntnis der Symbolwahrscheinlichkeiten. Sind diese sowohl dem Sender als auch dem Empfänger bekannt, so ist die mittlere Codewortlänge L_M oft nur unwesentlich größer als die Quellenentropie H.
- Insbesondere bei kleinen Dateien kann es zu Abweichungen zwischen den (erwarteten) Symbolwahrscheinlichkeiten p_μ und den (tatsächlichen) Symbolhäufigkeiten h_μ kommen. Besser wäre es hier, für jede Datei einen eigenen Huffman–Code zu generieren, der auf den tatsächlichen Gegebenheiten (h_μ) basiert.
- In diesem Fall muss aber dem Decoder auch der spezifische Huffman–Code mitgeteilt werden.

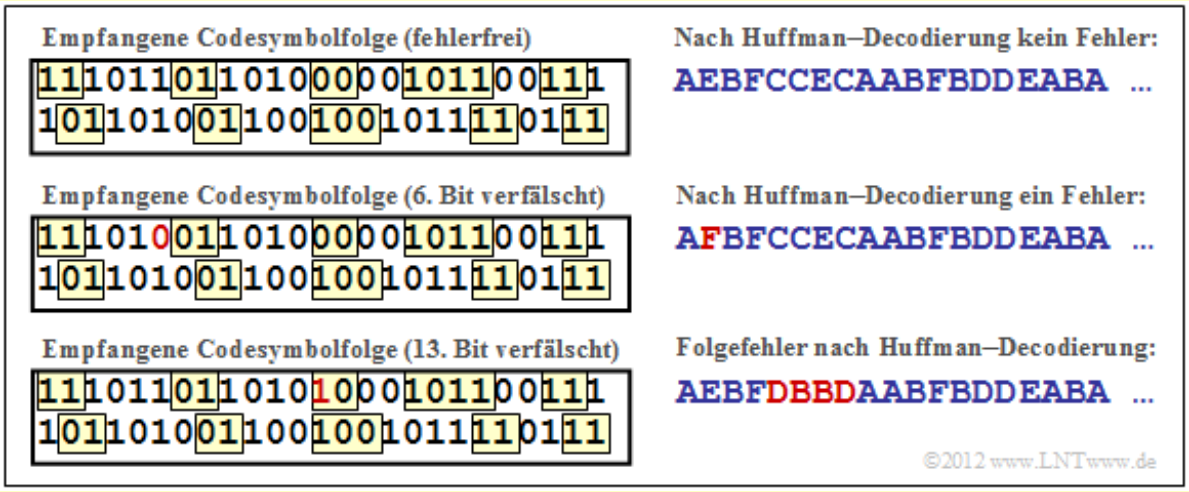
Dies führt zu einem gewissen *Overhead*, der nur wieder bei längeren Dateien vernachlässigt werden kann. Bei kleinen Dateien lohnt sich dieser Aufwand nicht.

Einfluss von Übertragungsfehlern auf die Decodierung (1)

Der Huffman–Code ist aufgrund der Eigenschaft „präfixfrei" verlustlos. Das bedeutet: Aus der binären Codesymbolfolge lässt sich die Quellensymbolfolge vollständig rekonstruieren. Kommt es aber bei der Übertragung zu einem Fehler (aus einer **0** wird eine **1** bzw. aus einer **1** eine **0**), so stimmt natürlich auch die Sinkensymbolfolge $\langle v_\nu \rangle$ nicht mit der Quellensymbolfolge $\langle q_\nu \rangle$ überein.

Die folgenden Beispiele zeigen, dass ein einziger Übertragungsfehler manchmal eine Vielzahl von Fehlern hinsichtlich des Ursprungstextes zur Folge haben kann.

Beispiel 1: Wir betrachten die gleiche Quellensymbolfolge und den gleichen Huffman–Code wie auf der **vorherigen Seite.** Die obere Grafik zeigt, dass bei fehlerfreier Übertragung aus der Binärfolge **111011** ... wieder die Folge **AEBFCC** ... rekonstruiert werden kann.



- Wird aber das 6. Bit verfälscht (von **1** auf **0**, rote Markierung in der mittlere Grafik), so wird aus dem Quellensymbol q_2 = **E** das Sinkensymbol v_2 = **F**.
- Eine Verfälschung von Bit 13 (von **0** auf **1**, rote Markierung in der unteren Grafik) führt dagegen zu einer Verfälschung von vier Quellensymbolen: **CCEC** ⇒ **DBBD**.

Die nächste Seite zeigt ein weiteres Beispiel zum Einfluss von Übertragungsfehlern bei Huffman.

Einfluss von Übertragungsfehlern auf die Decodierung (2)

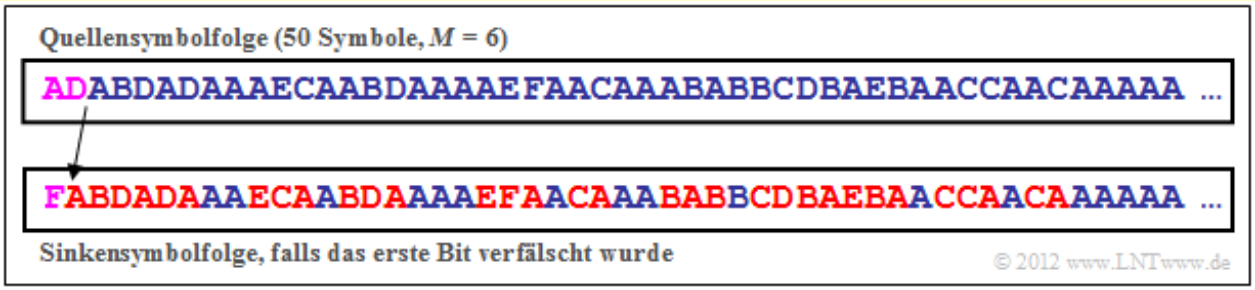
Beispiel 2: Eine zweite Nachrichtenquelle – ebenfalls mit Symbolumfang $M = 6$ – ist durch folgende Symbolwahrscheinlichkeiten gekennzeichnet:

$$p_{\rm A} = 0.50\,,\ p_{\rm B} = 0.19\,,\ p_{\rm C} = 0.11\,,\ p_{\rm D} = 0.09\,,\ p_{\rm E} = 0.06\,,\ p_{\rm F} = 0.05\,.$$

Hier führt der Huffman–Algorithmus zu folgender Zuordnung:

$$\mathbf{A} \rightarrow \mathbf{0},\ \mathbf{B} \rightarrow \mathbf{111},\ \mathbf{C} \rightarrow \mathbf{101},\ \mathbf{D} \rightarrow \mathbf{100},\ \mathbf{E} \rightarrow \mathbf{1101},\ \mathbf{F} \rightarrow \mathbf{1100}.$$

Die Quellensymbolfolge **ADABD** ... (siehe Grafik) wird somit durch die Codesymbolfolge **0'100'0'111'100'...** dargestellt. Die Hochkommata dienen hierbei nur der Orientierung.



Bei der Übertragung wird nun das erste Bit verfälscht: Anstelle von **01000111100...** empfängt man somit **11000111100...** Aus den beiden ersten Quellensymbolen **AD** → **0100** wird dann nach der Decodierung das Sinkensymbol **F** → **1100**. Die weiteren Symbole werden dann wieder richtig detektiert, aber nun nicht mehr beginnend bei $\nu = 3$, sondern bei $\nu = 2$.

Je nach Anwendung sind die Auswirkungen unterschiedlich:

- Handelt es sich bei der Quelle um einen natürlichen Text und bei der Sinke um einen Menschen, so bleibt der Großteil des Textes für den Leser verständlich.
- Ist die Sinke jedoch ein Automat, der sukzessive alle v_ν mit den entsprechenden q_ν vergleicht, so ergibt sich eine Verfälschungshäufigkeit von deutlich über 50%.
- Nur die blauen Symbole der Sinkensymbolfolge $\langle v_\nu \rangle$ stimmen dann (zufällig) mit den entsprechenden Quellensymbolen überein, während rote Symbole auf Fehler hinweisen.

Anwendung der Huffman–Codierung auf *k*–Tupel

Der Huffman–Algorithmus in seiner Grundform liefert dann unbefriedigende Ergebnisse, wenn

- eine Binärquelle ($M = 2$) vorliegt, zum Beispiel mit dem Symbolvorrat {**X**, **Y**},
- es statistische Bindungen zwischen den Symbolen der Eingangsfolge gibt,
- die Wahrscheinlichkeit des häufigsten Symbols deutlich größer ist als 50%.

Abhilfe schafft man in diesen Anwendungsfällen, in dem man mehrere Symbole zusammenfasst und den Huffman–Algorithmus auf einen neuen Symbolvorrat {**A**, **B**, **C**, **D**, ... } anwendet.

Bildet man k–Tupel, so steigt der Symbolumfang von M auf $M' = M^k$. Wir wollen im folgenden Beispiel die Vorgehensweise anhand einer Binärquelle ($M = 2$) verdeutlichen. Weitere Beispiele finden Sie in **Aufgabe A2.7, Aufgabe Z2.7** und **Aufgabe A2.8.**

Beispiel: Gegeben sei eine gedächtnislose Binärquelle ($M = 2$) mit den Symbolen {**X**, **Y**}:

- Die Symbolwahrscheinlichkeiten seien $p_X = 0.8$ und $p_Y = 0.2$.
- Damit ergibt sich die Quellenentropie zu $H = 0.722$ bit/Quellensymbol.
- Wir betrachten die Symbolfolge **XXXYXXXXXXXXYYXXXXXYYXXYXYXXYX ....**

Der Huffman–Algorithmus kann auf diese Quelle direkt nicht angewendet werden, das heißt, man benötigt ohne weitere Maßnahme für jedes binäre Quellensymbol auch ein Bit. Aber:

- Fasst man jeweils zwei binäre Symbole zu einem *Zweiertupel* ($k = 2$) entsprechend **XX** → **A**, **XY** → **B**, **YX** → **C**, **YY** → **D** zusammen, so kann man „Huffman" auf die resultierende Folge **ABAACADAABCBBAC ...** mit $M' = 4$ anwenden. Wegen

  $$p_A = 0.8^2 = 0.64, \quad p_B = 0.8 \cdot 0.2 = 0.16 = p_C, \quad p_D = 0.2^2 = 0.04$$

  erhält man **A** → **1**, **B** → **00**, **C** → **011**, **D** → **010** sowie

  $$L'_M = 0.64 \cdot 1 + 0.16 \cdot 2 + 0.16 \cdot 3 + 0.04 \cdot 3 = 1.56\,\text{bit/Zweiertupel}$$
  $$\Rightarrow \quad L_M = L'_M/2 = 0.78\,\text{bit/Quellensymbol}.$$

- Nun bilden wir Dreiertupel ($k = 3$). Mit den Kombinationen **XXX** → **A**, **XXY** → **B**, **XYX** → **C**, **XYY** → **D**, **YXX** → **E**, **YXY** → **F**, **YYX** → **G**, **YYY** → **H** kommt man für die oben angegebene Eingangsfolge zur äquivalenten Folge **AEBAGADBCC...** (basierend auf dem neuen Symbolumfang $M' = 8$) und zu folgenden Wahrscheinlichkeiten:

  $$p_A = 0.8^3 = 0.512, \quad p_B = p_C = p_E = 0.8^2 \cdot 0.2 = 0.128,$$
  $$p_D = p_F = p_G = 0.8 \cdot 0.2^2 = 0.032, \quad p_H = 0.2^3 = 0.008.$$

  Die Huffman–Codierung lautet somit: **A** → **1**, **B** → **011**, **C** → **010**, **D** → **00011**, **E** → **001**, **F** → **00010**, **G** → **00001**, **H** → **00000**. Damit erhält man für die mittlere Codewortlänge:

  $$L'_M = 0.512 \cdot 1 + 3 \cdot 0.128 \cdot 3 + (3 \cdot 0.032 + 0.008) \cdot 5 = 2.184\,\text{bit/Dreiertupel}$$
  $$\Rightarrow \quad L_M = L'_M/3 = 0.728\,\text{bit/Quellensymbol}.$$

  Bereits mit $k = 3$ wird also in diesem Beispiel die Quellenentropie $H = 0.722$ fast erreicht.

Der Shannon–Fano–Algorithmus (1)

Die Huffman–Codierung aus dem Jahr 1952 ist ein Sonderfall der **Entropiecodierung**. Dabei wird versucht, das Quellensymbol q_μ durch ein Codesymbol c_μ der Länge L_μ darzustellen, wobei folgende Konstruktionsvorschrift angestrebt wird:

$$L_\mu \approx \log_2\,(1/p_\mu)\,.$$

Da L_μ im Gegensatz zu $\log_2(1/p_\mu)$ ganzzahlig ist, gelingt dies nicht immer.

Bereits drei Jahre vor David A. Huffman haben **Claude E. Shannon** und **Robert Fano** einen ähnlichen Algorithmus angegeben, nämlich:

1. Man ordne die Quellensymbole nach fallenden Auftrittswahrscheinlichkeiten (identisch mit Huffman).
2. Man teile die sortierten Zeichen in zwei möglichst gleichwahrscheinliche Gruppen.
3. Der ersten Gruppe wird das Binärsymbol **1** zugeordnet, der zweiten die **0** (oder umgekehrt).
4. Sind in einer Gruppe mehr als ein Zeichen, so ist auf diese der Algorithmus rekursiv anzuwenden.

Beispiel 1: Wir gehen wie im **Einführungsbeispiel** für den Huffman–Algorithmus zu Beginn von Kapitel 2.3 von $M = 6$ Symbolen und den folgenden Wahrscheinlichkeiten aus:

$$p_{\rm A} = 0.30\,,\ p_{\rm B} = 0.24\,,\ p_{\rm C} = 0.20\,,\ p_{\rm D} = 0.12\,,\ p_{\rm E} = 0.10\,,\ p_{\rm F} = 0.04\,.$$

Dann lautet der Shannon–Fano–Algorithmus:

1. **AB** → **1**x (Wahrscheinlichkeit 0.54), **CDEF** → **0**x (Wahrscheinlichkeit 0.46),
2. **A → 11**, **B ⇒ 10**,
3. **C → 01**, (Wahrscheinlichkeit 0.20), **DEF** → **00**x, (Wahrscheinlichkeit 0.26),
4. **D → D**, (Wahrscheinlichkeit 0.12), **EF** → **000**x (Wahrscheinlichkeit 0.14),
5. **E → 0001**, **F → 0000**.

Anmerkung: Ein „x" weist wieder darauf hin, dass in nachfolgenden Codierschritten noch Bits hinzugefügt werden müssen.

Es ergibt sich hier zwar eine andere Zuordnung als bei der **Huffman–Codierung,** aber genau die gleiche mittlere Codewortlänge:

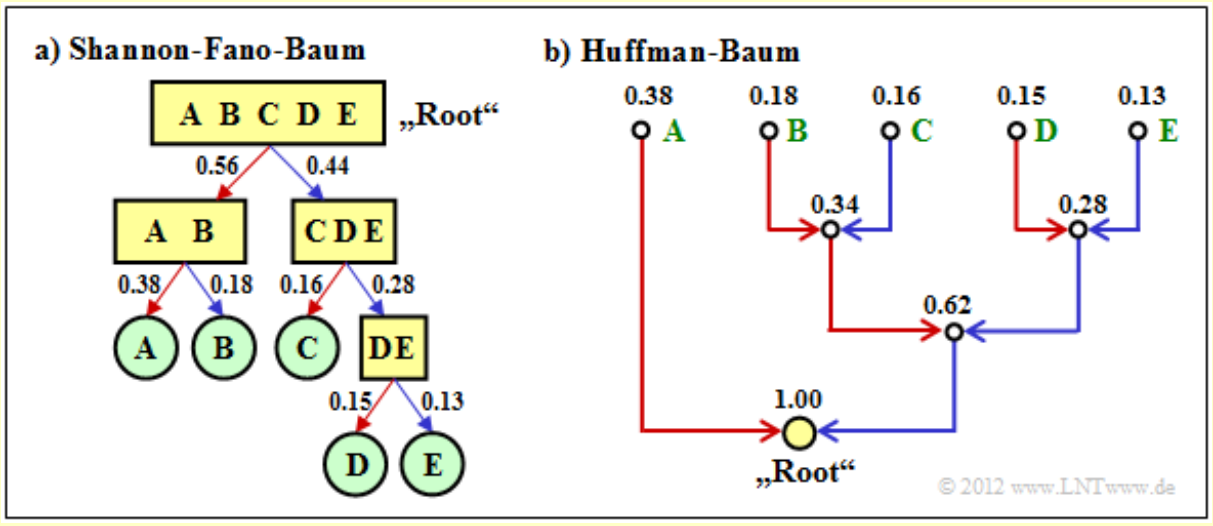
$$L_{\rm M} = (0.30 + 0.24 + 0.20)\cdot 2 + 0.12\cdot 3 + (0.10 + 0.04)\cdot 4 = 2.4\,{\rm bit/Quellensymbol}\,.$$

Der Shannon–Fano–Algorithmus (2)

Mit den Wahrscheinlichkeiten entsprechend dem **Beispiel 1** (auf der letzten Seite) führt der Shannon–Fano–Algorithmus zur gleichen mittleren Codewortlänge wie die Huffman–Codierung. Ebenso sind bei vielen (eigentlich: den meisten) anderen Wahrscheinlichkeitsprofilen Huffman und Shannon–Fano aus informationstheoretischer Sicht äquivalent. Es gibt aber durchaus Fälle, bei denen sich beide Verfahren hinsichtlich der (mittleren) Codewortlänge unterscheiden, wie das folgende Beispiel zeigt.

Beispiel 2: Wir betrachten $M = 5$ Symbole mit folgenden Wahrscheinlichkeiten:

$$p_{\rm A} = 0.38,\ \ p_{\rm B} = 0.18,\ \ p_{\rm C} = 0.16,\ \ p_{\rm D} = 0.15,\ \ p_{\rm E} = 0.13$$
$$\Rightarrow \quad H = 2.19\,\text{bit/Quellensymbol}.$$



Die Grafik zeigt die jeweiligen Codebäume für Shannon–Fano (links) bzw. Huffman (rechts). Die Ergebnisse lassen sich wie folgt zusammenfassen:

- Der **Shannon–Fano–Algorithmus** führt zum Code **A** → **11**, **B** → **10**, **C** → **01**, **D** → **001**, **E** → **000** und damit zur mittleren Codewortlänge

  $$L_{\rm M} = (0.38 + 0.18 + 0.16) \cdot 2 + (0.15 + 0.13) \cdot 3 = 2.28\ \text{bit/Quellensymbol}.$$

- Mit dem **Huffman–Algorithmus** erhält man **A** → **1**, **B** → **001**, **C** → **010**, **D** → **001** sowie **E** → **000** und eine etwas kleinere mittlere Codewortlänge:

  $$L_{\rm M} = 0.38 \cdot 1 + (1 - 0.38) \cdot 3 = 2.24\ \text{bit/Quellensymbol}.$$

- Es gibt keinen Satz von Wahrscheinlichkeiten, bei denen Shannon–Fano ein besseres Ergebnis liefert als der Huffman–Algorithmus, der den bestmöglichen Entropiecodierer bereitstellt.
- Die Grafik zeigt zudem, dass die Algorithmen im Baumdiagramm in unterschiedlichen Richtungen vorgehen, nämlich einmal von der Wurzel zu den Einzelsymbolen (Shannon–Fano), zum anderen von den Einzelsymbolen zur Wurzel (Huffman).

Arithmetische Codierung (1)

Eine weitere Form der Entropiecodierung ist die arithmetische Codierung. Auch bei dieser müssen die Symbolwahrscheinlichkeiten p_μ bekannt sein. Für den Index gelte weiter $\mu = 1, \text{ ... }, M$.

Hier nun ein kurzer Abriss über die Vorgehensweise:

- Im Gegensatz zur Huffman– und Shannon–Fano–Codierung wird bei arithmetischer Codierung eine Symbolfolge der Länge N gemeinsam codiert. Wir schreiben abkürzend $Q = \langle q_1, q_2, \text{ ... }, q_N \rangle$.
- Jeder solchen Symbolfolge Q_i wird ein reelles Zahlenintervall I_i zugewiesen, das durch den Beginn B_i und die Intervallbreite Δ_i gekennzeichnet ist.
- Der „Code" für die gesamte Folge Q_i ist die Binärdarstellung eines reellen Zahlenwertes aus diesem Intervall: $r_i \in I_i = [B_i, B_i + \Delta_i)$. Diese Notation besagt, dass zwar B_i zum Intervall I_i gehört (eckige Klammer), aber $B_i + \Delta_i$ gerade nicht mehr (runde Klammer).
- Es gilt stets $0 \le r_i < 1$. Sinnvollerweise wählt man r_i aus dem Intervall I_i derart, dass der Wert mit möglichst wenigen Bits darstellbar ist. Es gibt aber eine Mindestbitanzahl, die von der Intervallbreite Δ_i abhängt.

Der Algorithmus zur Bestimmung der Intervallparameter B_i und Δ_i wird auf der nächsten Seite an einem Beispiel erläutert, ebenso eine Decodiermöglichkeit. Zunächst folgt ein kurzes Beispiel zur Auswahl der reellen Zahl r_i in Hinblick auf minimale Bitanzahl. Genauere Informationen hierzu finden Sie zum Beispiel in **[BCK02]** und bei der Beschreibung zur **Aufgabe A1.12.**

Beispiel: Für folgende Parameter des arithmetischen Codieralgorithmus ergeben sich folgende reelle Ergebnisse r_i und folgende Codes, die zum zugehörigen Intervall I_i gehören:

- $B_i = 0.25, \Delta_i = 0.10 \Rightarrow I_i = [0.25, 0.35)$:

$$r_i = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 0.25 \quad \Rightarrow \quad \text{Code } \mathbf{01} \in I_i \,,$$

- $B_i = 0.65, \Delta_i = 0.10 \Rightarrow I_i = [0.65, 0.75)$; zu beachten: 0.75 gehört nicht zum Intervall:

$$r_i = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 0.6875 \quad \Rightarrow \quad \text{Code } \mathbf{1011} \in I_i \,.$$

Um den sequentiellen Ablauf zu organisieren, wählt man allerdings die Bitanzahl konstant zu

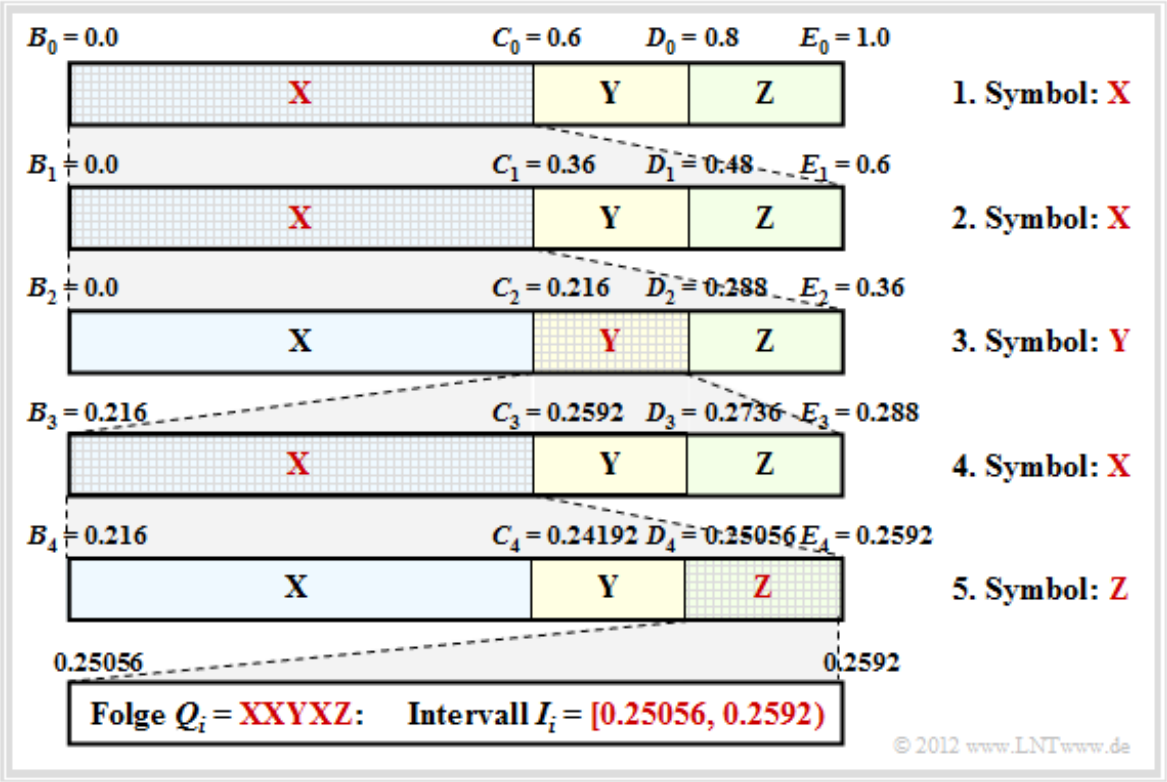
$$N_{\rm Bit} = \lceil \log_2 \, (1/\Delta_i) \rceil + 1 \,.$$

Mit der Intervallbreite $\Delta_i = 0.10$ ergibt sich $N_{\rm Bit} = 5$. Die tatsächlichen arithmetischen Codes wären also **01000** bzw. **10110**.

Arithmetische Codierung (2)

Die Aussagen der letzten Seite sollen nun an einem weiteren Beispiel verdeutlicht werden. Im Folgenden sei der Symbolumfang $M = 3$. Um Verwechslungen zu vermeiden, nennen wir die Symbole **X**, **Y** und **Z**:

- Übertragen werden soll die Zeichenfolge **XXYXZ** ⇒ Folgenlänge $N = 5$.
- Auszugehen ist von den Wahrscheinlichkeiten $p_X = 0.6$, $p_Y = 0.2$, $p_Z = 0.2$.



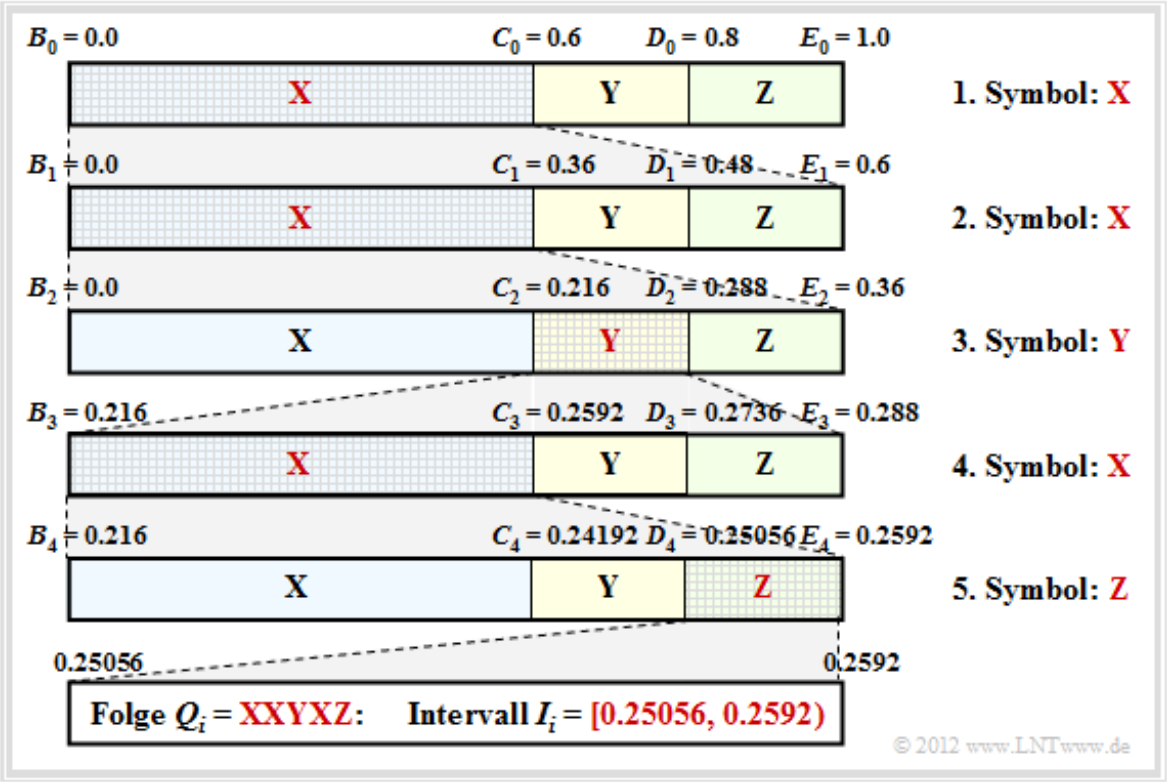
Die Grafik zeigt den Algorithmus zur Bestimmung der Intervallgrenzen.

- Man teilt zunächst den gesamten Wahrscheinlichkeitsbereich (zwischen 0 und 1) gemäß den Symbolwahrscheinlichkeiten p_X, p_Y und p_Z in drei Bereiche mit den Grenzen B_0, C_0, D_0 und E_0.
- Das erste Symbol ist **X**. Deshalb wird im nächsten Schritt der Wahrscheinlichkeitsbereich von $B_1 = B_0 = 0$ bis $E_1 = C_0 = 0.6$ wiederum im Verhältnis 0.6 : 0.2 : 0.2 aufgeteilt.
- Nach dem zweiten Symbol **X** liegen die Bereichsgrenzen bei $B_2 = 0$, $C_2 = 0.216$, $D_2 = 0.288$ und $E_2 = 0.36$. Da nun das Symbol **Y** ansteht, erfolgt die Unterteilung des Bereiches 0.216 ... 0.288.
- Nach dem fünften Symbol **Z** liegt das Intervall I_i für die betrachtete Symbolfolge Q_i = **XXYXZ** fest. Es muss nun eine reelle Zahl r_i gefunden werden, für die gilt: $0.25056 \le r_i < 0.2592$.
- Die einzige reelle Zahl im Intervall $I_i = [0.25056, 0.2592)$, die man mit 7 Bit darstellen kann, ist $r_i = 1 \cdot 2^{-2} + 1 \cdot 2^{-7} = 0.2578125$. Damit liegt die Coderausgabe fest: **0100001**.

Das Ergebnis dieser Codierung wird auf der nächsten Seit interpretiert.

Arithmetische Codierung (3)

Auf der letzten Seite wurde die arithmetische Codierung an einem Beispiel verdeutlicht. Bevor wir das Ergebnis interpretieren, zunächst nochmals die Vorgehensweise als Grafik:



Für diese $N = 5$ Symbole werden also 7 Bit benötigt, genau so viele wie bei Huffman–Codierung mit der Zuordnung **X** → **1**, **Y** → **00**, **Z** → **01**. Die arithmetische Codierung ist allerdings dann dem Huffman–Code überlegen, wenn die tatsächlich bei Huffman verwendete Bitanzahl noch mehr von der optimalen Verteilung abweicht, zum Beispiel, wenn ein Zeichen extrem häufig vorkommt.

Oft wird aber die Intervallmitte – im Beispiel 0.25488 – binär dargestellt: 0.01000010011 .... Die Bitanzahl erhält man daraus mit $\Delta_5 = 0.2592 - 0.25056 = 0.00864$ wie folgt:

$$N_{\rm Bit} = \left\lceil \log_2 \frac{1}{0.00864} \right\rceil + 1 = \lceil \log_2 \ 115.7 \rceil + 1 = 8 \, .$$

Damit lautet der arithmetische Code für dieses Beispiel mit $N = 5$ Eingangszeichen: **01000010**.

Der Decodiervorgang lässt sich ebenfalls anhand der obigen Grafik erklären. Die ankommende Bitsequenz **0100001** wird zu $r = 0.2578125$ gewandelt. Dieser liegt im ersten und zweiten Schritt jeweils im ersten Bereich ⇒ Symbol **X**, im dritten Schritt in zweiten Bereich ⇒ Symbol **Y**, usw.

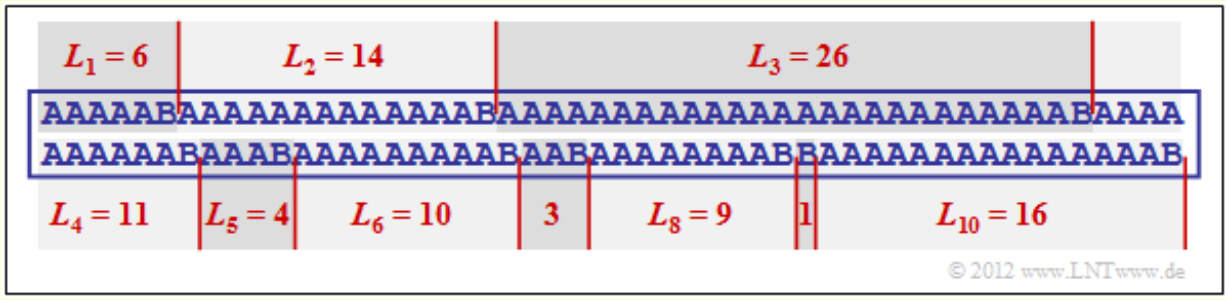
Weitere Informationen zu diesem Thema finden Sie in **WIKIPEDIA** sowie in **[BCK02].**

Lauflängencodierung – Run–Length Coding

Wir betrachten eine Binärquelle ($M = 2$) mit dem Symbolvorrat {**A**, **B**}, wobei ein Symbol sehr viel häufiger auftritt als das andere. Beispielsweise sei p_A sehr viel größer als p_B.

Eine Entropiecodierung macht hier nur dann Sinn, wenn man diese auf ***k*–Tupel** anwendet. Eine zweite Möglichkeit bietet die **Lauflängencodierung** (englisch: *Run–Length Coding*, RLC), die das seltenere Zeichen **B** als Trennzeichen betrachtet und die Längen L_i der einzelnen Substrings als Ergebnis liefert.

Beispiel: Die Grafik zeigt eine beispielhafte Binärfolge mit den Wahrscheinlichkeiten $p_A = 0.9$ und $p_B = 0.1$, woraus sich die Quellenentropie $H = 0.469$ bit/Quellensymbol ergibt. Die Beispielfolge der Länge $N = 100$ beinhaltet genau zehnmal das Symbol **B** und neunzigmal das Symbol **A**, das heißt, die relativen Häufigkeiten stimmen exakt mit den Wahrscheinlichkeiten überein.



Man erkennt an diesem Beispiel:

- Die Binärfolge hat die Länge $N = 100$. Die Lauflängencodierung dieser Folge ergibt in Dezimalschreibweise die Folge 6, 14, 26, 11, 4, 10, 3, 9, 1, 16.
- Stellt man die Längen L_1, ... , L_{10} mit jeweils 5 Bit dar, so benötigt man $5 \cdot 10 = 50$ Bit. Die Datenkomprimierung ist nicht viel schlechter als der theoretische Grenzwert, der sich durch die Quellenentropie H ergibt ($H \cdot N \approx 47$ Bit).
- Die direkte Anwendung einer Entropiecodierung – zum Beispiel nach **Huffman** – hätte hier keine Datenkomprimierung zur Folge; man benötigt weiterhin 100 Bit. Auch bei der Bildung von Dreiertupeln würde man mit Huffman noch mehr Bit benötigen als durch RLC, nämlich 54 Bit.

Das Beispiel zeigt aber auch zwei Probleme der Lauflängencodierung:

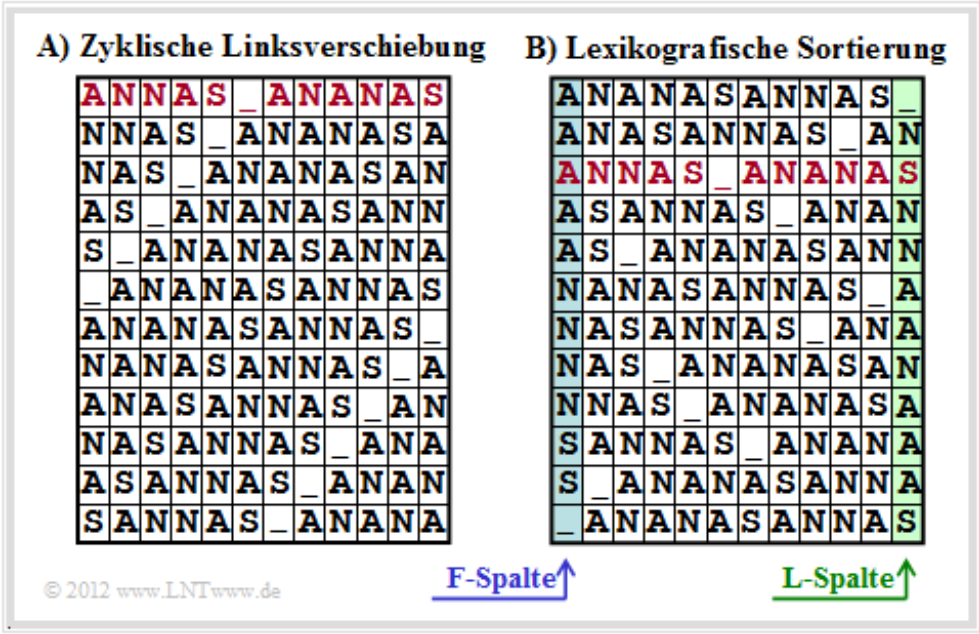
- Die Längen L_i der Substrings sind nicht begrenzt. Hier muss man besondere Maßnahmen treffen, wenn eine Länge L_i größer ist als $2^5 = 32$ (falls $N_{Bit} = 5$), zum Beispiel die Variante *Run–Length Limited Coding* (RLLC). Siehe auch **[Meck09]** und **Aufgabe A2.13.**
- Endet die Folge nicht mit einem **B** – was bei kleiner Wahrscheinlichkeit p_B eher der Normalfall ist, so muss auch für das Dateiende eine Sonderbehandlung vorgesehen werden.

Burrows–Wheeler–Transformation (1)

Zum Abschluss dieses Quellencodier–Kapitels behandeln wir noch kurz den 1994 von Michael Burrows und **David J. Wheeler** veröffentlichten Algorithmus **[BW94],**

- der zwar alleine keinerlei Komprimierungspotenzial besitzt,
- aber die Komprimierungsfähigkeit anderer Verfahren stark verbessert.

Die Burrows–Wheeler–Transformation bewerkstelligt eine blockweise Sortierung von Daten, die in der folgenden Grafik am Beispiel des Textes **ANNAS_ANANAS** der Länge $N = 12$ verdeutlicht werden:



- Zunächst wird aus dem String der Länge N eine $N \times N$–Matrix erzeugt, wobei sich jede Zeile aus der Vorgängerzeile durch zyklische Linksverschiebung ergibt.
- Danach wird die BWT–Matrix lexikografisch sortiert. Das Ergebnis der Transformation ist die letzte Spalte ⇒ L–Spalte. Im Beispiel ergibt sich der String **_NSNNAANAAAS**.
- Des Weiteren muss auch der Primärindex I weitergegeben werden. Dieser gibt die Zeile der sortierten BWT–Matrix an, die den Originaltext enthält (in der Grafik rot markiert).

Zur Bestimmung von L–Spalte und Primärindex I sind natürlich keine Matrixoperationen erforderlich. Vielmehr findet man das BWT–Ergebnis mit Zeigertechnik sehr schnell.

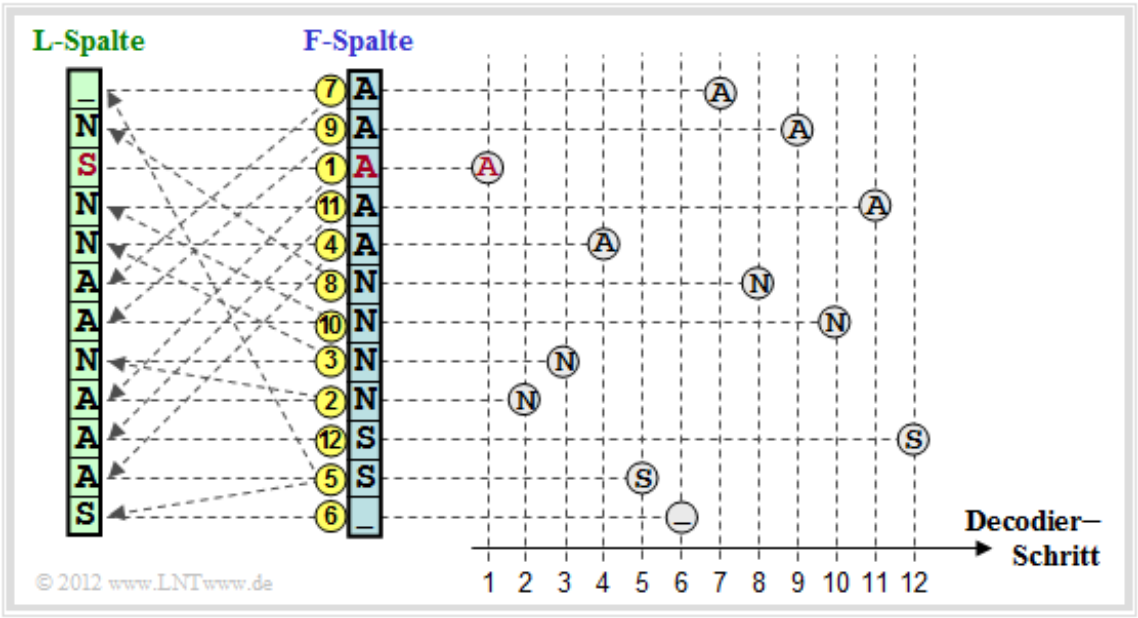
Außerdem ist zum BWT–Verfahren noch anzumerken:

- Ohne Zusatzmaßnahme ⇒ eine nachgeschaltete „echte Kompression" – führt die BWT zu keiner Datenkomprimierung: Vielmehr ergibt sich sogar eine geringfügige Erhöhung der Datenmenge, da außer den N Zeichen nun auch der Primärindex I übermittelt werden muss.
- Bei längeren Texten ist dieser Effekt aber vernachlässigbar. Geht man von 8 Bit–ASCII–Zeichen (jeweils ein Byte) und der Blocklänge $N = 256$ aus, so erhöht sich die Byte–Anzahl pro Block nur von 256 auf 257, also lediglich um 0.4%.

Wir verweisen auf die ausführlichen Beschreibungen zur BWT in **[Abel03b].**

Burrows–Wheeler–Transformation (2)

Abschließend soll noch dargestellt werden, wie der Ursprungstext aus der L–Spalte der BWT–Matrix rekonstruiert werden kann. Dazu benötigt man noch den Primärindex I, sowie die erste Spalte der BWT–Matrix. Diese F–Spalte (von „First") muss nicht übertragen werden, sondern ergibt sich aus der L–Spalte sehr einfach durch lexikografische Sortierung.



Die Grafik zeigt die Vorgehensweise für das betrachtete Beispiel:

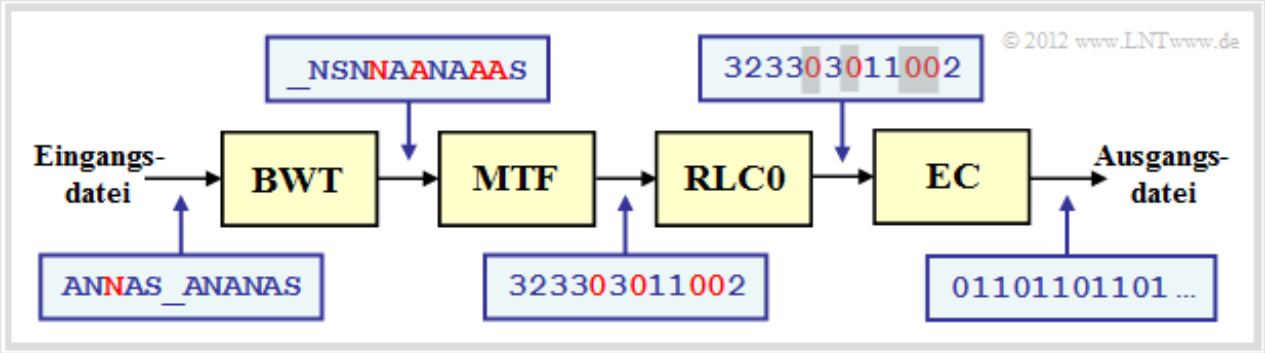
- Man beginnt in der Zeile mit dem Primärindex I. Als erstes Zeichen wird das rot markierte **A** in der F–Spalte ausgegeben. Dieser Schritt ist in der Grafik mit einer gelben **(1)** gekennzeichnet.
- Dieses **A** ist das dritte **A**–Zeichen in der F–Spalte. Man sucht nun das dritte **A** in der L–Spalte, findet dieses in der mit **(2)** markierten Zeile und gibt das zugehörige **N** der F–Spalte aus.
- Das letzte **N** der L–Spalte findet man in der mit **(3)** gekennzeichneten Zeile. Ausgegeben wird das Zeichen der F–Spalte in der gleichen Zeile, also wieder ein **N**:

Nach $N = 12$ Decodierschritten ist die Rekonstruktion abgeschlossen. Dieses Beispiel hat gezeigt, dass die BWT nichts anderes ist als ein Sortieralgorithmus für Texte.

Das Besondere daran ist, dass die Sortierung eindeutig umkehrbar ist. Diese Eigenschaft und zusätzlich seine innere Struktur sind die Grundlage dafür, dass man das BWT–Ergebnis mittels bekannter und effizienter Verfahren wie **Huffman** (eine Form der Entropiecodierung) und **RLC** (*Run–Length Coding*) komprimieren kann.

Anwendungsszenario für BWT

Als Beispiel für die Einbettung der **Burrows–Wheeler–Transformation** (BWT) in eine Kette von Quellencodierverfahren wählen wir eine in **[Abel03a]** vorgeschlagene Struktur:



Wir verwenden dabei das gleiche Textbeispiel **ANNAS_ANANAS** wie auf der letzten Seite. Die entsprechenden Strings nach den einzelnen Blöcken sind in der Grafik ebenfalls angegeben.

- Das Ergebnis der **BWT** lautet: **_NSNNAANAAAS**. An der Textlänge $N = 12$ hat die BWT nichts verändert, doch gibt es jetzt vier Zeichen, die identisch mit ihren Vorgängerzeichen sind (in der Grafik rot hervorgehoben). Im Originaltext war dies nur einmal der Fall.
- Im nächsten Block **MTF** (*Move–To–Front*) wird aus jedem Eingangszeichen aus der Menge {**A**, **N**, **S**, **_**} ein Index $I \in \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$. Es handelt sich hierbei aber nicht um ein einfaches *Mapping*, sondern um einen Algorithmus, der in **Aufgabe Z1.14** angegeben ist.
- Für unser Beispiel lautet die MTF–Ausgangsfolge **323303011002**, ebenfalls mit der Länge $N = 12$. Die vier Nullen in der MTF–Folge (in der Grafik ebenfalls mit roter Schrift) geben an, dass an diesen Stellen das BWT–Zeichen jeweils gleich ist wie sein Vorgänger.
- Bei großen ASCII–Dateien kann die Häufigkeit der **0** durchaus >50% betragen, während die anderen 255 Indizes nur selten auftreten. Zur Komprimierung einer solchen Textstruktur eignet sich eine **Lauflängencodierung** (englisch: *Run–Length Coding*, RLC) hervorragend.
- Der Block **RLC0** in obiger Codierungskette bezeichnet eine spezielle Lauflängencodierung für Nullen. Die graue Schattierung der Nullen soll andeuten, dass hier eine lange Nullsequenz durch eine spezifische Bitfolge (kürzer als die Nullsequenz) maskiert wurde.
- Der Entropiecodierer (**EC**, z.B. Huffman) sorgt für eine weitere Komprimierung. BWT und MTF haben in der Codierungskette nur die Aufgabe, durch eine Zeichenvorverarbeitung die Effizienz von RLC0 und EC zu steigern. Die Ausgangsdatei ist wieder binär.