Datenkompression:

Arithmetische Codierung

H. Fernau

email: fernau@uni-trier.de

SoSe 2011

Universität Trier

Erinnerung

- Für möglichst gute Codierungen kann man o.B.d.A. *Präfix-codes* betrachten.

- Die Entropie liefert eine theoretische Schranke für die mögliche verlustfreie Kompression einer Quelle mit vorgegebener Statistik.

- Huffman-Codes liefern Codierungen “nahe an der Entropie”.

- Noch besser sind erweiterte Huffman-Codes.
  Nachteil: Das zu betrachtende Alphabet wir schnell riesig groß.

Arithmetische Codes

Beispiel 1 *Erweiterter Huffman-Algorithmus: für ein Quellenalphabet mit* 256 *Symbolen und für Folgen der Länge* $m = 3$ *muss der Algorithmus*

16 777 216 *(!)*

neue Symbole betrachten.

Arithmetische Codes (Grundschema): Für gegebenen Text

$$u_1u_2u_3u_4u_5u_6u_7u_8u_9\ldots u_m \in \Sigma^m,$$

– berechne eine numerische Repräsentation $\in [0,1)$ und
– für diese Repräsentation berechne den binären Code.

Folge der Symbole $\longrightarrow$ numerische Repräsentation $\longrightarrow$ binärer Code.

Dabei ist m (impliziter) *Parameter* der arithmetischen Codierung.

Erzeugen der numerischen Repräsentation Sei $\Sigma = \{a_1, a_2, a_3, \ldots, a_n\}$ mit

$$P(a_1), \ldots, P(a_n).$$

Wir definieren $F(0) = 0$ und für $i = 1, 2, \ldots, n$

$$F(i) = \sum_{k=1}^{i} P(a_k).$$

Sei $\boxed{a_{i_1} a_{i_2} a_{i_3} \ldots a_{i_m}}$ eine Folge.
Eine *numerische Repräsentation* für *diese* Folge ist (irgend)ein Bruch in

$$[l^{(m)}, u^{(m)}).$$

Definiere

$$l^{(1)} = F(i_1 - 1) \qquad u^{(1)} = F(i_1)$$

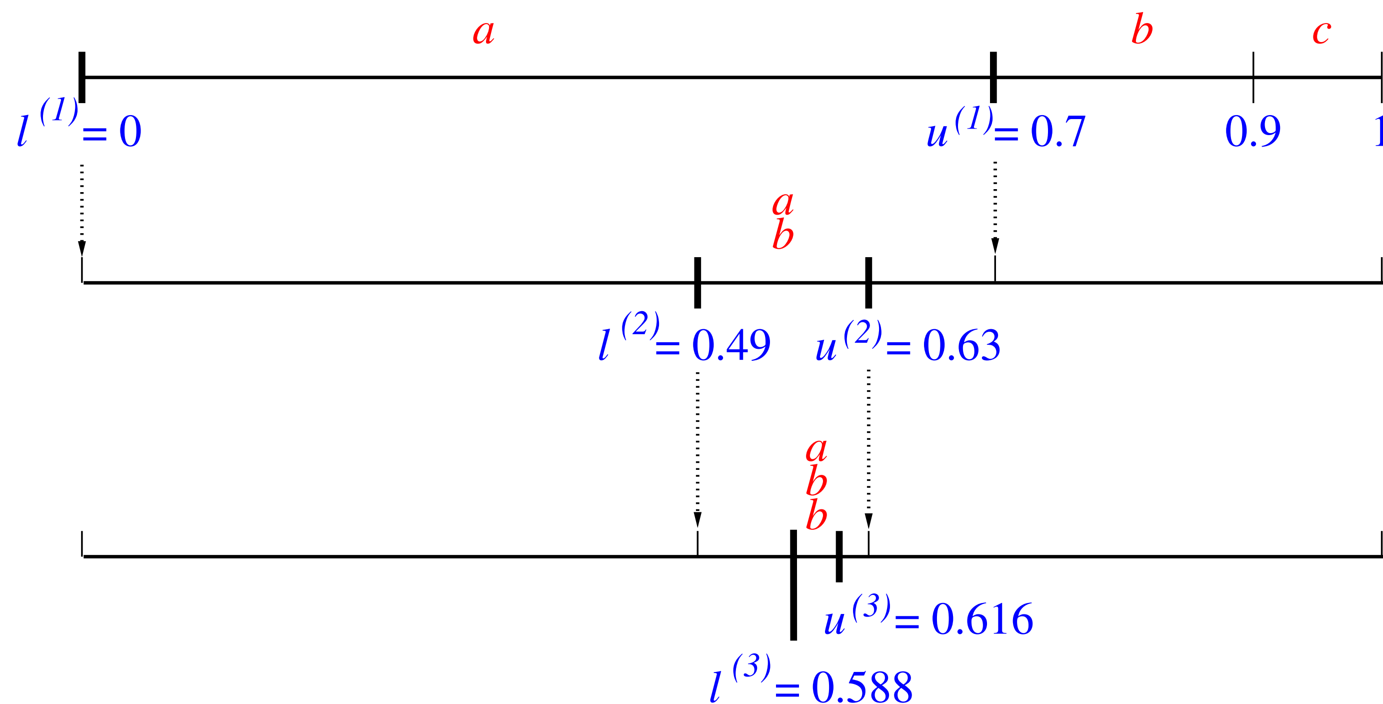
und für alle $k = 2, 3, \ldots, m$

$$\begin{aligned} l^{(k)} &= l^{(k-1)} + \left(u^{(k-1)} - l^{(k-1)}\right) F(i_k - 1) \\ u^{(k)} &= l^{(k-1)} + \left(u^{(k-1)} - l^{(k-1)}\right) F(i_k). \end{aligned}$$

Beispiel 2

$\Sigma = \{a, b, c\}$ *mit* $P(a) = 0.7, P(b) = 0.2, P(c) = 0.1$ *und* $m = 3$.

Das Intervall z.B. für die Folge abb *ist* $[0.588, 0.616)$*:*

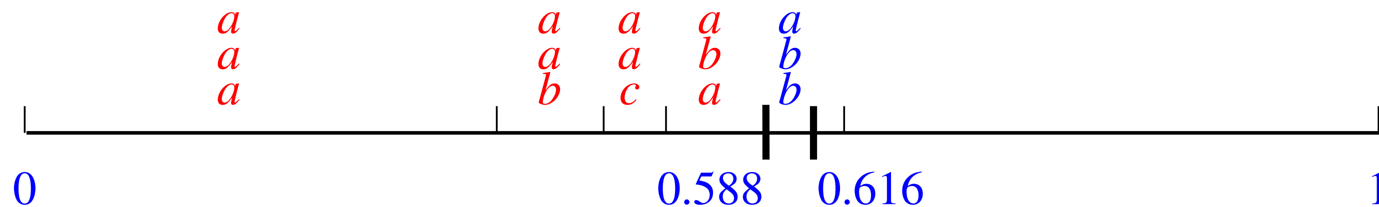


Übliche numerische Repräsentation: Intervallmittelpunkt

$$\overline{T}(a_{i_1}a_{i_2}\ldots a_{i_m}) = \frac{u^{(m)} + l^{(m)}}{2},$$

Bemerkungen:

– Für $x = a_{i_1}a_{i_2}\ldots a_{i_m}$, $\overline{T}(x) = \sum_{y <_{lex} x} P(y) + P(x)/2$,
wobei $P(x) = P(a_{i_1}) \cdot P(a_{i_2}) \ldots P(a_{i_m})$.
$<_{lex}$ ordnet Wörter zuerst nach Länge, dann wie im Wörterbuch.

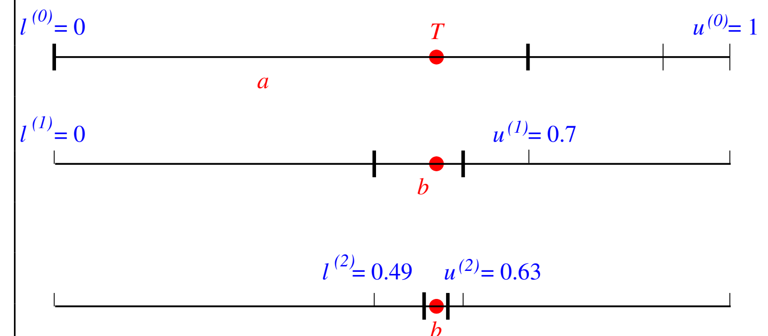


– Die einzige Information, die der Algorithmus braucht, ist die Funktion F.

Decodierung

Decodierung: *numerische Repräsentation* T $\rightarrow$ *Folge* x *für* T.

```
Sei l^(0) = 0 und u^(0) = 1;
for k := 1 to m do
begin
 T* := (T − l^(k−1))/(u^(k−1) − l^(k−1));
 Finde i_k, sodass
     F(i_k − 1) ≤ T* < F(i_k);
 Return(a_{i_k});
 Aktualisiere l^(k) und u^(k);
end.
```



Binärer Code Nehmen wir an, dass

$$\boxed{\overline{T}(a_{i_1}a_{i_2}\ldots a_{i_m}) = (u^{(m)} + l^{(m)})/2}$$

Es sei, wie früher,

$$P(a_{i_1}a_{i_2}\ldots a_{i_m}) = P(a_{i_1}) \cdot P(a_{i_2}) \ldots P(a_{i_m}).$$

Wir definieren

$$\boxed{\ell(a_{i_1}a_{i_2}\ldots a_{i_m}) \;=\; \left\lceil \log \frac{1}{P(a_{i_1}a_{i_2}\ldots a_{i_m})} \right\rceil + 1.}$$

Der *binäre Code der Repräsentation*:

$$\ell(a_{i_1}a_{i_2}\ldots a_{i_m}) \text{ höchstwertige Bits der Zahl } \overline{T}(a_{i_1}a_{i_2}\ldots a_{i_m}).$$

Beispiel 3

Sei $\Sigma = \{a, b\}$, $P(a) = 0.9$, $P(b) = 0.1$ *und der Parameter* $m = 2$.

x	$\overline{T}(x)$	binär	$\ell(x)$	Code
aa	0.405	0.0110011110...	2	01
ab	0.855	0.1101101011...	5	11011
ba	0.945	0.1111000111...	5	11110
bb	0.995	0.1111111010...	8	11111110

Erinnerung: Shannon-Fano-Codierung

$\lfloor y \rfloor_\ell$: Abschneiden einer Binärzahl y nach ℓ-tem Nachkomma-Bit

Ziel: Binär-Code aus möglichst wenigen Nachkomma-Bits.

Satz 4 *Sei* $\left[l^{(m)}, u^{(m)}\right)$ *ein Intervall für die Folge* $x \in \Sigma^m$. $\rightsquigarrow$

$$\boxed{\left\lfloor \overline{T}(x) \right\rfloor_{\ell(x)} \in \left[l^{(m)}, u^{(m)}\right).}$$

Beweis Es gilt: $u^{(m)} - l^{(m)} = P(x)$. Deshalb ist

$$\overline{T}(x) = l^{(m)} + \frac{P(x)}{2}.$$

Dann bekommen wir:

$$\overline{T}(x) \geq l^{(m)} + \frac{1}{2^{\ell(x)}}.$$

$$\boxed{\begin{array}{rcl} P(x)/2 & \geq & 1/2^{\ell(x)} \\ 2^{\ell(x)} & \geq & 2/P(x) \\ \ell(x) & \geq & \log(2/P(x)) \\ \ell(x) & \geq & \log(1/P(x)) + 1 \end{array}}$$

Wenn wir also den Bruch $\overline{T}(x)$ hinter dem $\ell(x)$-ten Bit abschneiden, dann gilt

$$\left\lfloor \overline{T}(x) \right\rfloor_{\ell(x)} \geq l^{(m)}.$$

Satz 5 *Für beliebige $x, y \in \Sigma^m$ sind die entsprechenden arithmetischen Codes $C(x)$ und $C(y)$ präfixfrei, d.h., weder ist $C(x)$ Präfix von $C(y)$ noch umgekehrt.*

Beweis Es sei $a \in [0, 1)$ mit

$$a = 0.b_1 b_2 \dots b_n.$$

Es gilt: $b = 0.b_1 b_2 \dots b_n c_{n+1} c_{n+2} \dots$ genau dann, wenn

$$b \in [a, a + \frac{1}{2^n}).$$

Wir beweisen: $\left[\left\lfloor \overline{T}(x) \right\rfloor_{\ell(x)}, \left\lfloor \overline{T}(x) \right\rfloor_{\ell(x)} + \frac{1}{2^{\ell(x)}}\right)$ liegt in $\left[l^{(m)}, u^{(m)}\right)$.

$$\begin{aligned} u^{(m)} - \left\lfloor \overline{T}(x) \right\rfloor_{\ell(x)} &\geq u^{(m)} - \overline{T}(x) \\ &= \frac{P(x)}{2} \geq \frac{1}{2^{\ell(x)}}. \end{aligned}$$

□

Wie gut ist arithmetische Codierung ?

Sei x eine Folge, $\ell(x) = \left\lceil \log \frac{1}{P(x)} \right\rceil + 1$ Bits. Es sei L_{A^m} die erwartete Länge für einen arithmetischen Code mit Parameter m. Dann gilt:

$$\begin{aligned} L_{A^m} &= \sum_{x \in \Sigma^m} P(x)\ell(x) \\ &\leq -\sum_{x \in \Sigma^m} P(x) \log P(x) + 2 \sum_{x \in \Sigma^m} P(x) \\ &= H(\Sigma^m) + 2. \end{aligned}$$

Weil L_{A^m} immer größer als die Entropie ist, bekommen wir

$$H(\Sigma^m) \leq L_{A^m} \leq H(\Sigma^m) + 2.$$

Für die durchschnittliche Länge L_A pro Symbol bekommen wir

$$\frac{H(\Sigma^m)}{m} \leq L_A \leq \frac{H(\Sigma^m)}{m} + \frac{2}{m}$$

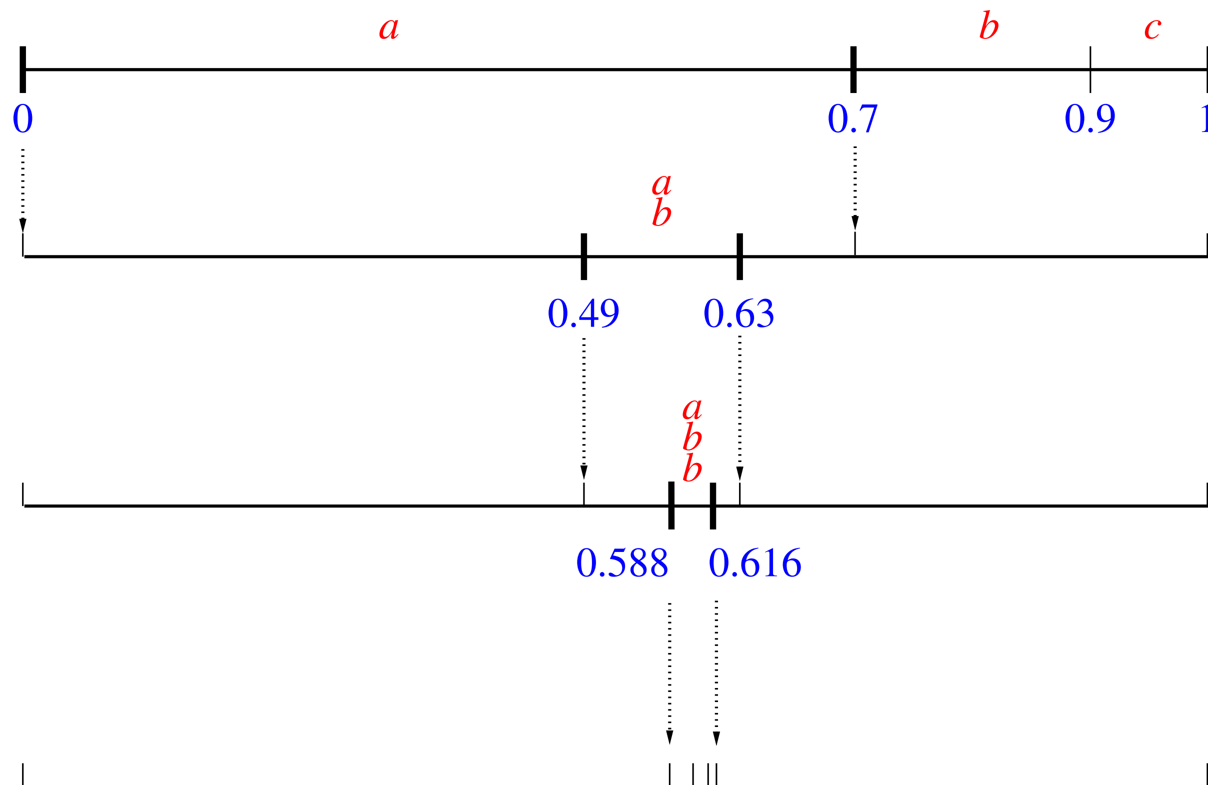
und weil $H(\Sigma^m) = mH(\Sigma)$, gilt

$$\boxed{H(\Sigma) \leq L_A \leq H(\Sigma) + \tfrac{2}{m}.}$$

Implementierung der Arithmetische Codierung

Schwierigkeit: Präzision!

Die Länge der Intervalle konvergiert gegen Null.



Re-Skalierung als Lösung

ganz einfache Umrechnungsformeln:

1. $I = [l, u) \subseteq [0, 1/2) \leadsto l = 2l;\ u = 2u$
2. $I = [l, u) \subseteq [1/2, 1) \leadsto l = 2(l - 0.5);\ u = 2(u - 0.5)$
3. $I = [l, u) \subseteq [1/4, 3/4) \leadsto l = 2(l - 0.25);\ u = 2(u - 0.25)$

Es werden also “möglichst große” Teilintervalle von $[0, 1)$ gespeichert.
Dieser Trick ist auch in anderen Situationen mit numerischen Problemen anwendbar.

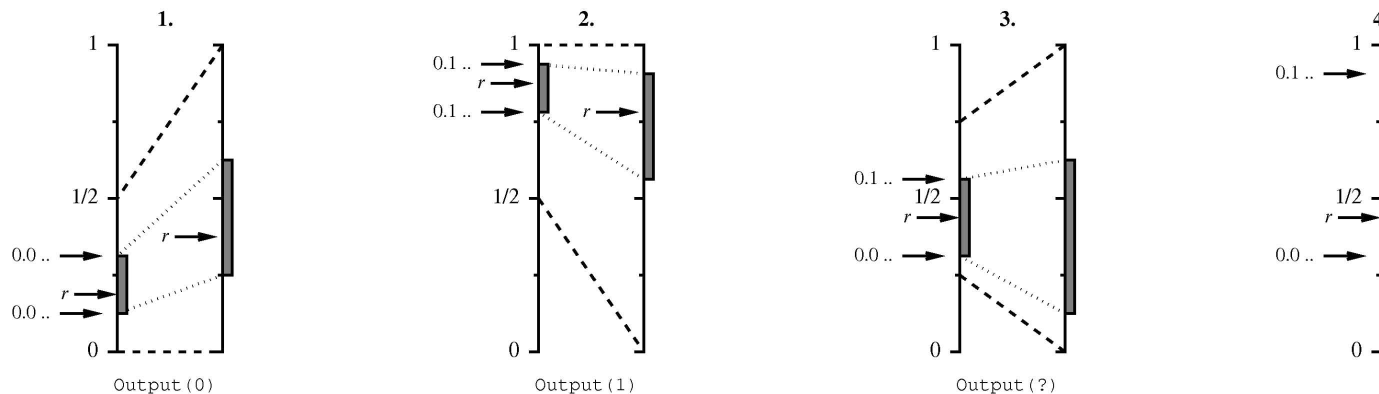
Implementierung der Arithmetische Codierung

Schwierigkeit: Rechengenauigkeit (die Intervall-Länge konvergiert gegen 0).

Lösung: Re-Skalierung der Intervalle I, wenn

1. $I \subseteq [0, 1/2)$ oder 2. $I \subseteq [1/2, 1)$ oder 3. $I \subseteq [1/4, 3/4)$.

4. Fall: keiner der drei Fälle tritt ein $\rightarrow$ keine Re-Skalierung.



Die ersten beiden Fälle sind klar:

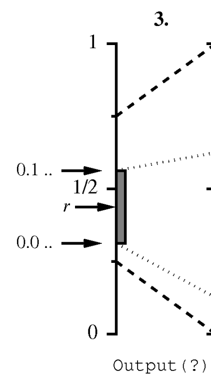
Das höchste Bit ist festgelegt, es wird ausgegeben und das Intervall wird reskaliert.

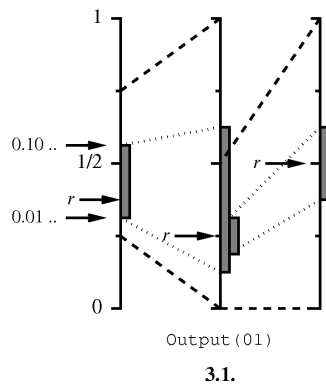
Im dritten und vierten Fall muss die Ausgabe warten.

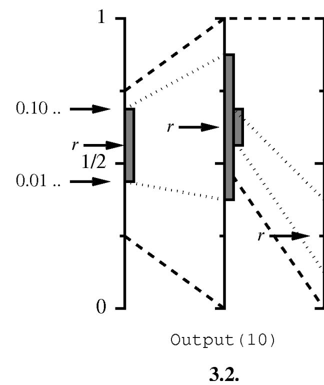
Beispiel 6 *Arithmetische Codierung mit Reskalieren*
Sei $\Sigma = \{a, b, c\}$, $P(a) = 0.8$, $P(b) = 0.02$, $P(c) = 0.18$.
Wir wollen $acba$ *codieren.*
1. Zeichen ist $a \rightsquigarrow$ *Intervall* $[0, 0.8) \rightsquigarrow$ *Fall 4 (keine Ausgabe).*
2. Zeichen ist $c \rightsquigarrow$ *Intervall* $[0.656, 0.8) \rightsquigarrow$ *Fall 2*
$\Longrightarrow$ *Ausgabe 1, Reskalierung zu Intervall* $[(2*(0.656-0.5), 2*(0.8-0.5)) = [0.312, 0.6)$.
3. Zeichen ist $b \rightsquigarrow$ *Intervall* $[0.5424, 0.54816) \rightsquigarrow$ *Fall 2*
$\Longrightarrow$ *Ausgabe 1, Reskalierung zu Intervall* $[(2*(0.5424-0.5), 2*(0.54816-0.5)) = [0.0848, 0.09632)$.
Dies neue Intervall liegt vollständig in $[0, 0.5)$, *sodass wir erneut reskalieren können (Fall 1).*
$\Longrightarrow$ *Ausgabe 0, Reskalierung zu Intervall* $[(2*(0.0848), 2*(0.09632)) = [0.1696, 0.19264)$.
Erneut Fall 1 $\Longrightarrow$ *Ausgabe 0, Reskalierung zu Intervall* $[0.3392, 0.38528)$.
Erneut Fall 1 $\Longrightarrow$ *Ausgabe 0, Reskalierung zu Intervall* $[0.6784, 0.77056)$.
Jetzt Fall 2 $\Longrightarrow$ *Ausgabe 1, Reskalierung zu Intervall* $[0.3568, 0.54112)$.
4. Zeichen ist $a \rightsquigarrow$ *Intervall* $[0.3568, 0.504256) \rightsquigarrow$ *Fall 3.*
Wenn dies das letzte Zeichen ist, müssen dem Empfänger noch einige weitere Bits aus dem Intervall gesendet werden, z.B. 100... *für die Zahl* 0.5.

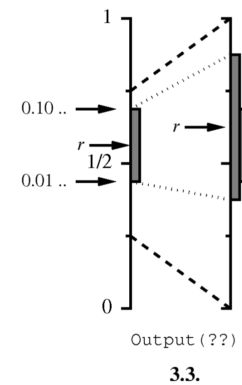
Implementierung der Arithmetische Codierung

Skalierung der Intervalle I wenn $I \subseteq [1/4, 3/4)$ (Fall 3.).









Algorithmus

Codiere das k-te Symbol a_{i_k};

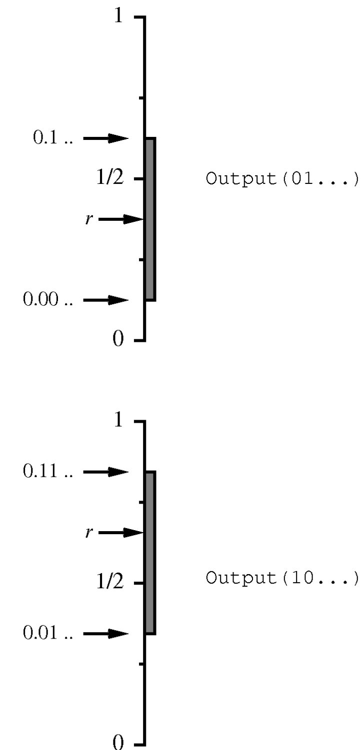
/* am Anfang $l := 0$; $u := 1$; $I := [l, u)$; bits_to_follow=0 */

$u = l + |I| \cdot F(i_k)$; $l = l + |I| \cdot F(i_{k-1})$; $I := [l, u)$;

```
while(1) {
   if I ⊆ [0,0.5) {
      output(0);
      while (bits_to_follow > 0)
         { output(1); bits_to_follow -- ; }
      Re-Skalierung [0,0.5) → [0,1) :  l = 2l;  u = 2u;   }
   else if I ⊆ [0.5,1) {
      output(1);
      while (bits_to_follow > 0)
         { output(0); bits_to_follow -- ; }
      Re-Skalierung [0.5,1) → [0,1) :  l = 2(l − 0.5);  u = 2(u − 0.5); }
   else if I ⊆ [0.25,0.75) {
      bits_to_follow ++ ;
      Re-Skalierung [0.25,0.75) → [0,1) :  l = 2(l−0.25);  u = 2(u−0.25);
}
   else break;
}  /* while(1) */
```

Ende der Codierung;

```
bits_to_follow ++ ;
if (l^(m) < 0.25) {
 output(0);
 while (bits_to_follow > 0)
  { output(1); bits_to_follows -- ; };
}
else {
 output(1);
 while (bits_to_follow > 0)
  { output(0); bits_to_follows -- ; };
}
```



Decodierung

Der Decodierer muss “genauso” wie der Codierer arbeiten.
Dabei: Problem mit Terminierung.

Lösung 1: Die Länge der letzten Zeichenfolge ist zusätzlich festzulegen, oder
Lösung 2: Ein ausgezeichnetes Endezeichen ist zu übertragen.

Außerdem wird zuallererst der Parameter m übertragen (d.h., die Länge der codierten Zeichenfolge) oder aber das Verfahren legt m von vornherein konstant fest. Der Decodierer muss dann also genau m Zeichen ausgeben (oder weniger, falls Endezeichen bekannt).

Betrachten wir die Arbeitsweise des Decodierers wiederum an einem Beispiel.

Beispiel 7 *Arithmetische Decodierung mit Reskalieren*

Sei $\Sigma = \{a, b, c\}$, $P(a) = 0.8$, $P(b) = 0.02$, $P(c) = 0.18$.

Wir wollen 1100011000...0 *decodieren.*

Die entsprechende Binärzahl liegt im Intervall $[0, 0.8) \rightsquigarrow$ *1. Zeichen* a.
Das Intervall liegt nicht in $[0, 0.5)$ *oder in* $[0.5, 1) \rightsquigarrow$ *keine Reskalierung.*
Die entsprechende Binärzahl liegt in den letzten 18 % vom Intervall $[0, 0.8)$ $\rightsquigarrow$ *2. Zeichen* c.
Das so gefundene Intervall $[0.625, 0.8)$ *liegt in* $[0.5, 1) \rightsquigarrow$ *Reskalierung.*
$\rightsquigarrow$ *neues Intervall* $[0.312, 0.6)$; *neues Binärwort (Rest nach Löschen des höchstwertigen Bits):* 100011000...0.
Wir finden: Keine weitere Reskalierung möglich; nächstes Zeichen b.
Das neue Intervall ist $[0.5424, 0.54816) \rightsquigarrow$ *Reskalierung(en) möglich:*
fünfmal (!) Reskalieren $\rightsquigarrow$ *Intervall* $[0.3568, 0.54112)$;
Löschen der fünf höchstwertigen Bits: $\rightsquigarrow$ 1000...0.
Dem entspricht als letztes Zeichen a.

Weitere Modifikationen: Ganzzahlige Implementierung.

$$[0,1) \subset \mathbb{R} \quad \to \quad \{0,1,2,\ldots,M\}$$

Wir benutzen auch ganzzahlige Zähler $g_i =$ '$\#a_i$ in Text x'.
Sei: $G(i) = \sum_{j=1}^{i} g_i$. Statt

$$[0,F(1)),\ [F(1),F(2)),\ldots[F(n-1),1)$$

benutzen wir:

$$[0,G(1)),\ [G(1),G(2)),\ldots[G(n-1),M).$$

Das i-te Intervall des Intervalls $[l,u)$ definieren wir wie folgt:

$$[l+(u-l+1)(G(i-1)\div G(n)),\ l+(u-l+1)(G(i)\div G(n))].$$

In der Praxis wird $M = 2^m - 1$ gewählt (Binärwörter fixer Länge).

Ganzzahlige Implementierung (Witten et al.)

```
/* ARITHMETIC ENCODING ALGORITHM. */

/* Call encode_symbol repeatedly for each symbol in the message. */
/* Ensure that a distinguished ‘terminator’ symbol is encoded last, then */
/* transmit any value in the range [low, high). */

encode_symbol(symbol, cum-freq)
 range = high - low
 high = low + range*cum_freq[symbol-1]
 low = low + range*cum_freq[symbol]
```

```
/* ARITHMETIC DECODING ALGORITHM. */

/* ‘Value’ is the number that has been received. */
/* Continue calling decode_symbol until the terminator symbol is returned. */

decode_symbol(cum-freq)
 find symbol such that
  cum_freq[symbol] <= (value-low)/(high-low) < cum_freq[symbol-1]
      /* This ensures that value lies will within the new */
      /* [low, high) range that will be calculated by */
      /* the following lines of code. */  range = high - low
 high = low + range*cum_freq[symbol-1]
 low = low + range*cum_freq[symbol]
 return symbol
```

Adaptive Arithmetische Codierung

Problem: Quellenstatistik oft unbekannt.
Lösung (wiederum): Adaption.

Ausgangsannahme: Alle Zeichen sind gleichwahrscheinlich.
$\rightsquigarrow$ Anfangszählerstand überall gleich Eins.
Mit der augenblicklichen Statistik wird sodann immer das nächste Zeichen codiert.
Hierbei werden auch evtl. Reskalierungen vorgenommen.
Anschließend wird die Statistik aktualisiert.
Diese Reihenfolge ist wichtig, da der Decodierer dieselbe Statistik verwenden muss.

Patentrechte Compression FAQ (kohalik)

http://www.faqs.org/faqs/compression-faq/part1/section-7.html

What about patents on data compression algorithms?

IBM holds many patents on arithmetic coding (United States Patents: 4,122,440 4,286,256 4,295,125 4,463,342 4,467,317 4,633,490 4,652,856 4,792,954 4,891,643 4,901,363 4,905,297 4,933,883 4,935,882 5,045,852 5,099,440 5,142,283 5,210,536 5,414,423 5,546,080).
It has patented in particular the Q-coder implementation of arithmetic coding.
The JBIG standard, and the arithmetic coding option of the JPEG standard requires use of the patented algorithm.
No JPEG-compatible method is possible without infringing the patent, because what IBM actually claims rights to is the underlying probability model (the heart of an arithmetic coder).

See also below details on many other patents on arithmetic coding (4,973,961 4,989,000 5,023,611 5,025,258 5,272,478 5,307,062 5,309,381 5,311,177 5,363,099 5,404,140 5,406,282 5,418,532). The list is not exhaustive.

Ausblick

Bislang: Annahme, aufeinanderfolgende Zeichen seien unabhängig voneinander
$\leadsto$ aus “einfachen” Statistiken kann man solche für Wörter der Länge m ausrechnen

Diese Annahme ist in der Praxis falsch.

Im Folgenden:
Techniken, die diese Abhängigkeiten ausdrücklich ausnutzen.

Lauflängencodierung

Oft ist sogar das Gegenteil richtig:

Bei Schwarz-Weiß-Bildern folgt auf ein schwarzes Pixel mit hoher Wahrscheinlichkeit ein schwarzes, während auf ein weißes mit hoher Wahrscheinlichkeit ein weißes folgt.
Bei der Lauflängencodierung von Schwarz-Weiß-Bildern werden nur die Längen der Folgen von (notwendigerweise abwechselnden) weißen und schwarzen Pixeln übertragen (o.E. wird als Erstes von einer weißen Pixelfolge evtl. der Länge Null ausgegangen).
Gibt es mehr als zwei verschiedene Ausgangszeichen, so muss außer der Lauflänge auch noch die Zeichenart selbst übertragen werden.

Eine ähnliche Technik wird z.B. beim Fascimile-Standard (Fax-Geräte) eingesetzt.

Patentrechte: Fast zum Schmunzeln, wenn's nicht traurig wäre

```
Run length encoding

Tsukiyama has two patents on run length encoding: 4,586,027 and 4,872,009 grantec
in 1986 and 1989 respectively. The first one covers run length encoding in
its most primitive form: a length byte followed by the repeated byte. The second
patent covers the 'invention' of limiting the run length to 16 bytes and thus
the encoding of the length on 4 bits. Here is the start of claim 1 of patent
4,872,009, just for pleasure:

1. A method of transforming an input data string comprising a plurality of
data bytes, said plurality including portions of a plurality of consecutive
data bytes identical to one another, wherein said data bytes may be of a pluralit
of types, each type representing different information, said method comprising
the steps of: [...]
```

siehe http://www.faqs.org/faqs/compression-faq/part1/section-7.html

Hinweis

Datenkompression muss nicht immer “einfach” sein.

Lauflängencodierung im Zweidimensionalen würde z.B. bedeuten, eine $\{0,1\}$-Matrix mit möglichst wenigen “Rechtecken” zu beschreiben, deren Fläche insgesamt alle Koordinaten von Einsen in der Matrix abdeckt (und nur diese).

Dieses Optimierungsproblem ist NP-hart.

Ist k eine Schranke auf die zulässige Zahl von Rechtecken, so ist das so parameterisierte Entscheidungsproblem in der Klasse FPT. $\leadsto$ Vorlesung parameterisierte Algorithmen

An unserem Lehrstuhl beschäftigen wir uns (u.a.) mit solchen “parameterisierten Algorithmen”.
Für das vorgestellte Problem wurde in einer Diplomarbeit bei uns ein $O^*(2^k)$-Algorithmus entwickelt.
Wir würden uns natürlich eine Implementierung wünschen. . .

Literatur

1. K. Sayood. *Introduction to Data Compression*, Morgan Kaufmann Publishers, Inc., 3. Auflage, 2006.
2. I.H. Witten, R.M. Neal and J.G. Cleary, *Arithmetic coding for data compression*, Commun. Assoc. Comput. Mach., 30(6), (1987) pp 520–540.
3. D. Salomon. *Data Compression; The Complete Reference,* Springer, 1998.