Multimedia

§4 Lossless Compression

Prof. Dr. Georg Umlauf

Hochschule Konstanz
University of Applied Sciences

Content

§4.1 Run-length coding

- Run-Length Encoding (RLE)

Properties:

- Makes use of multiple repetitions of identical symbols.
- Very fast.
- Uses very little resources.
- Works well for graphics
  - Part of BMP, JPEG, TIFF and PCX

§4.1 Run-length coding

Example

- Symbol sequence

  AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD

  can be compressed to

  4A3B2A5B8C1D1A1B1C1B3A4B3C1D

  which means

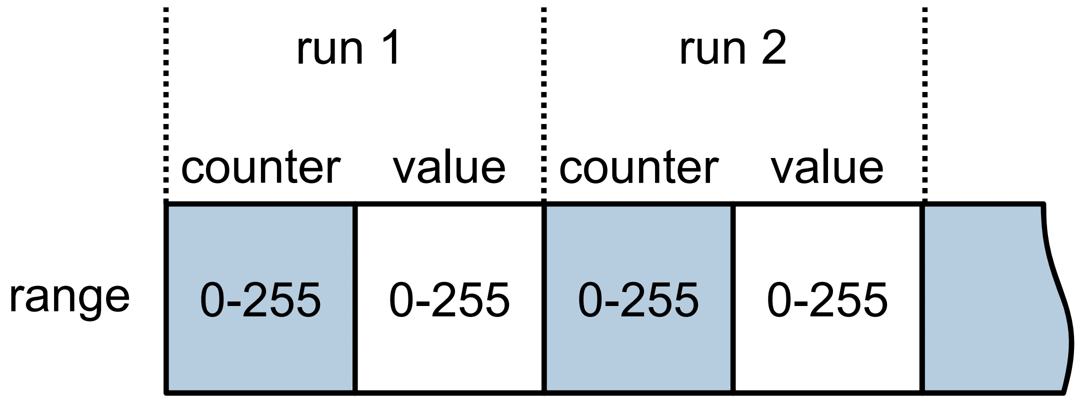
  4-times **A**, 3-times **B**, 2-times **A**, 5-timeds **B**, etc.

  - Compression rate: 38/28 ≈1,357...
  - Saving: 26% $\left(= 1 - \frac{1}{\text{compression rate}} \right)$

§4.1 Run-length coding

8-Bit RLE (1)

- Code consist of 2-byte blocks
- Each block codes one **run**, i.e. a sequence of identical symbols.
  - First block (**counter**) stores the **run-length**, i.e. the length of the sequence.
  - Second block (**value**) stores the **run-value**, i.e. the symbol that is repeated.



§4.1 Run-length coding

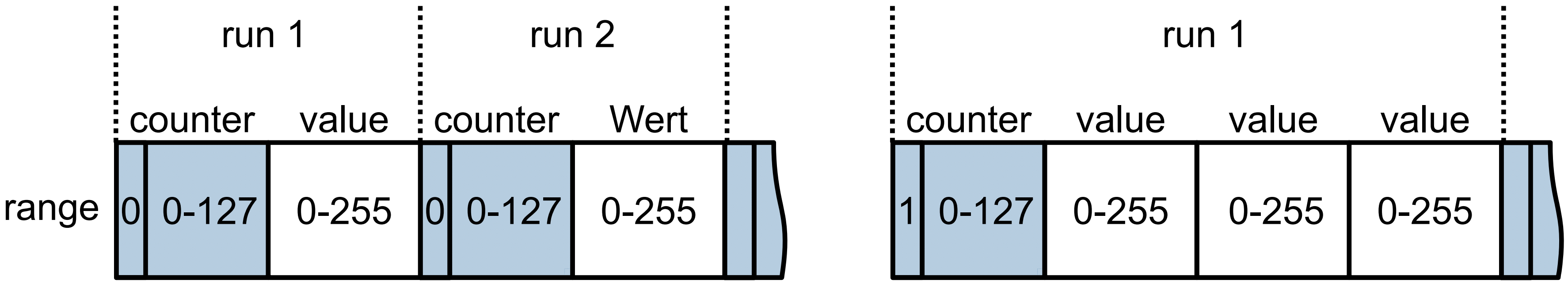
8-Bit RLE (2)

- Properties:
  - Length of runs is in the range of 1 to 256 symbols, since length 0 is not used.
  - Runs of length >256 are split into several parts.
  - For runs of length 1, the data size is doubled.
  - ➡ In the worst case this means a doubling of the total data volume.

§4.1 Run-length coding

8-Bit RLE Variant (1)

- Allow also un-coded runs.
- Most significant bit of counter marks un-coded runs:
  - 0 ➡ coded run ➡ second block stores a run-value
  - 1 ➡ un-coded run ➡ subsequent bytes store an un-coded run
- Lower 7 Bit of the counter store for
  - a coded run ➡ length of the run
  - an un-coded run ➡ number of un-coded symbols



8-Bit RLE Variant (2)

- Properties:
    - Length of runs is in the range of 1 to 128 symbols, since length 0 is not used.
    - Runs of length >128 are split into several parts.
    - Well suited for data with many runs of length 1.
    - Works well e.g. for images with 8-bit color depth.

§4.1 Run-length coding

Example

- Symbol sequence

  AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD

  compressed with 8-bit RLE

  4A3B2A5B8C1D1A1B1C1B3A4B3C1D

  and compressed with 8-Bit RLE variant

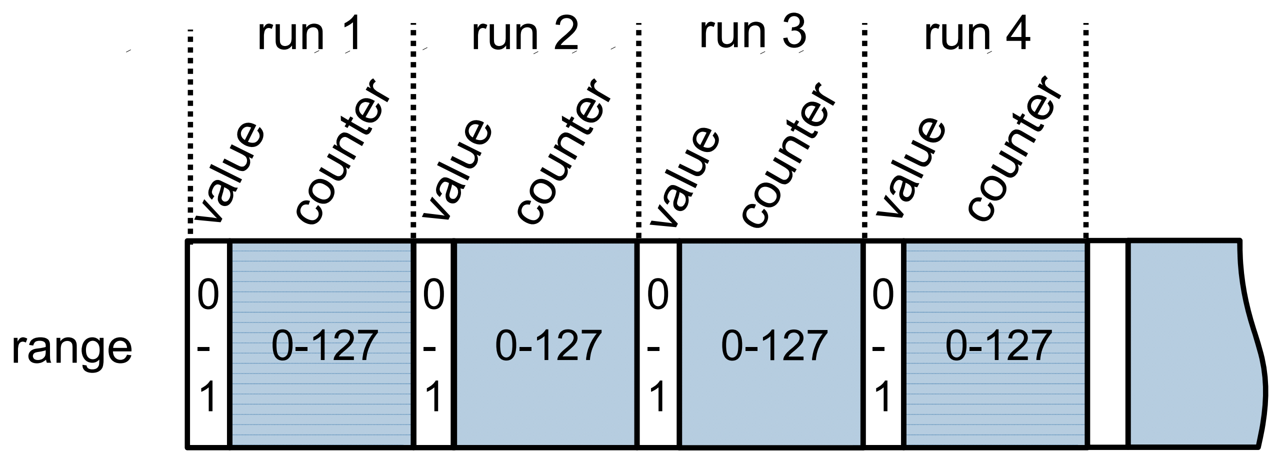
  4A3B2A5B8C<u>5</u>DABCB3A4B3C1D

  - Compression rate: 38/28 ≈1,357... rsp. 38/24 ≈1,583...
  - Saving: 26,3% rsp. 36,8%

§4.1 Run-length coding

1-Bit RLE (1)

- Code consists of 1-byte blocks:
  - Most significant bit stores the value.
  - Lower 7 bits store the counter.
- Works well e.g. for mono-chrome graphics, binary data, etc.



Content

Discrete memoryless source (DMS)

- **Source:** Sender of a message or producer of information.
- **Discrete:** Only a finite number of unique symbols from a so-called **alphabet U** is necessary to represent the message.
  - Example:
    - U is the alphabet of Latin letters of ASCII-symbols,
    - $U = \{0,1\}$ for binary data,...
- **Memoryless:** The occurrence of a symbol does not depend on the occurrence of other symbols.

➡ **Source without memory**

Problem

- **Given:** A DMS
  - with alphabet $U = \{u_1, \dots, u_n\}$ of symbols and
  - their probability to occur $P_U(u_i) = p_i, u_i \in U$.
- **Wanted:** A code
  - for the symbols u_i to binary code words w_i,
  - that minimizes the length of the binary code for any sequence of symbols from U and
  - allows for a perfect de-coding.

Example: Message of 66 symbols and entropy 2,66.

Symbol	a	e	i	s	t	space	newline
Frequency	10	15	11	7	9	12	2

§4.2 Entropy coding

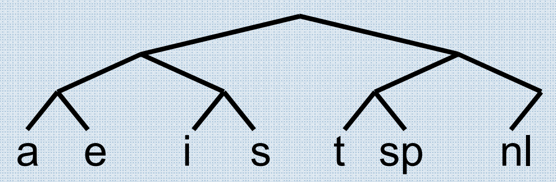
Code trees

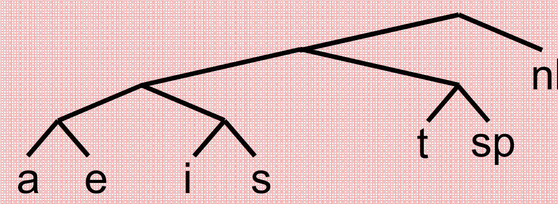
- Restriction to binary codes for the symbols, where the encoding scheme can be represented by a code tree.
- Symbols are given in the leaves of the code tree.

Example

Binary code with constant bit length

Symbol	Code
a	000
e	001
i	010
s	011
t	100
sp	101
nl	110





Symbol	Code
a	0000
e	0001
i	0010
s	0011
t	010
sp	011
nl	1

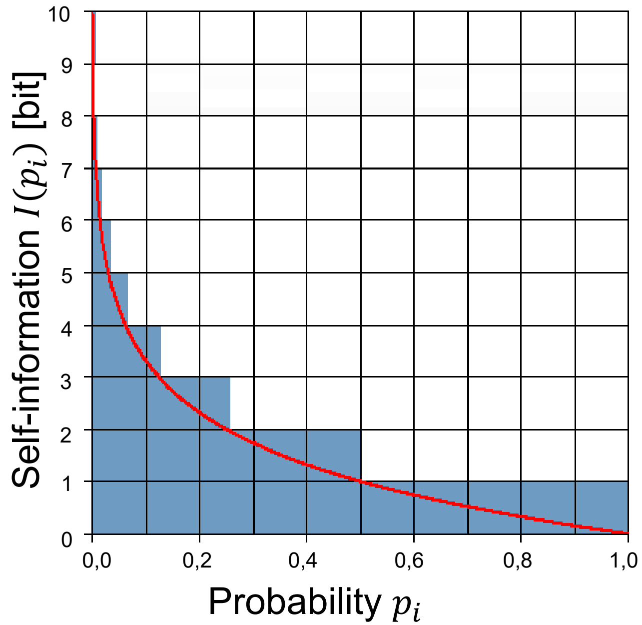
Binary code with variable bit length

Entropy

- Which binary code is more efficient – constant or variable bit length?
- How many bits are at least necessary to code all symbols, in order to
  - distinguish all symbols uniquely from each other and
  - compress the message with minimal size?

- **Entropy** is a measure for the information content of a source:
  - Infrequent symbols have a larger **self-information** $I(p_i) := -\log_2(p_i)$ than frequent ones.
  - **Entropy:** $H = \sum_i p_i \cdot I(p_i)$.

    Expected value of the self-information.



Entropy coding

- **Entropy coding:** Use binary code with variable bit length, where frequent symbols u_i get a shorter code word w_i.
- **Average code word length** is a measure for the efficiency of a coding

  $L = \sum_i p_i \cdot \ell_i$, where ℓ_i is the length [bit] of w_i.
- **Redundancy:** $(L - H)/H$.
- **Examples:** Shannon code, Shannon-Fano code, Huffman code
- **Problem:** How to distinguish the symbols from each other?
  - It can happen that a short code word is part (e.g. beginning) of another code word.
  - Solution: delimiter symbol? ➡ NO!

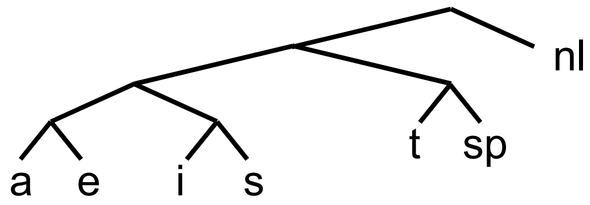
Fano condition (prefix-condition)

- A code satisfies the Fano condition, if there is no code word of a symbol that is the prefix (beginning) of the code word for another symbol.

  ➡ **prefix-free code**

- Codes, that can be represented by code trees, always satisfy the Fano condition.
- Codes, that satisfy Fano condition, can be de-coded uniquely.

Example



Symbol	Code
a	0000
e	0001
i	0010
s	0011
t	010
sp	011
nl	1

- De-code the following binary sequence using the above code tree: 0010110110100010

Kraft inequality

- **Goal:** Construct for source coding a code with variable bit length code words that can be de-coded, i.e. which is prefix-free.

➡ **Question:** Given a certain combination of code word lengths $\ell_1, \ell_2, \dots, \ell_n$, does a corresponding prefix-free code exist?

➡ **Answer: Kraft-(McMillan) inequality**

A binary prefix-free code with code word lengths $\ell_1, \ell_2, \dots, \ell_n$ exists, if and only if (iff)

$$\sum_{i=1}^{n} 2^{-\ell_i} \leq 1.$$

Prefix-free codes

- The proof for the Kraft inequality yields an algorithm to construct a prefix-free code from given code word lengths:

```
(1)   Sort the code word lengths ascending ℓ_1 ≤ ℓ_2 ≤ ... ≤ ℓ_n.
(2)   Construct a complete binary tree with height
      h = ℓ_n  and set  i = 1.
(3)   Choose an arbitrary knot of depth ℓ_i, that has not yet
      been processed, and prune the tree.
(4a)  If i = n,     stop,
(4b)  otherwise,  increase i by one and go back to (2).
```

Shannon code

- **Idea:** Construct a prefix-free code, such that a symbol u_i with frequency p_i, is coded using as many bits as given by the self-information $I(u_i) = -\log_2 p_i$, i.e. $\ell_i = \lceil -\log_2 p_i \rceil$.

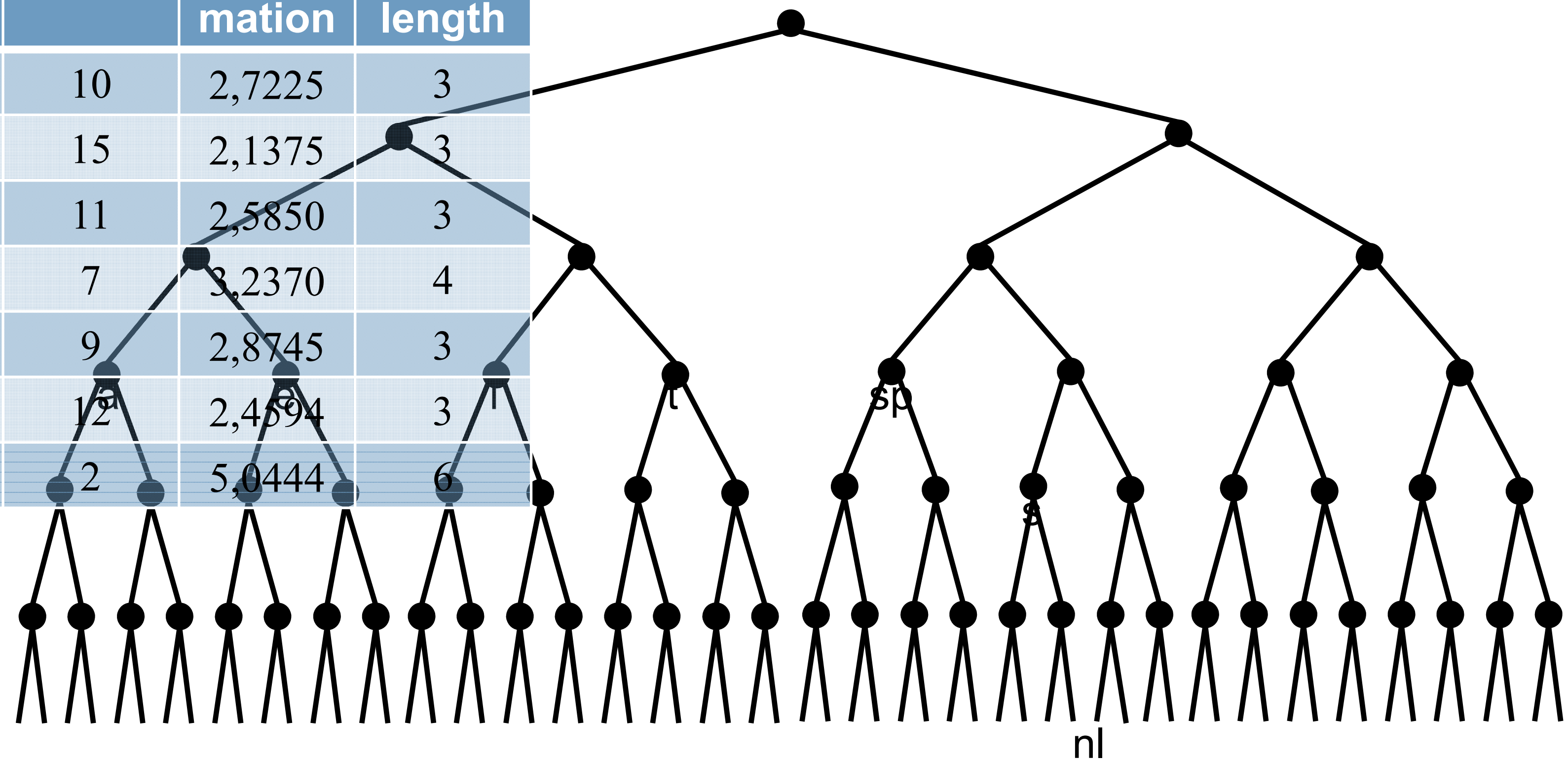
➡ For this idea the Kraft inequality is satisfied, i.e. such a code exists.

➡ For the average code word length $L(U)$ of a Shannon code for the alphabet U with entropy $H(U)$ this yields:

$$H(U) \leq L(U) \leq H(U) + 1.$$

Example

Symbol	Fre-quency	Self-infor-mation	Code word length
a	10	2,7225	3
e	15	2,1375	3
i	11	2,5850	3
s	7	3,2370	4
t	9	2,8745	3
sp	12	2,4594	3
nl	2	5,0444	6

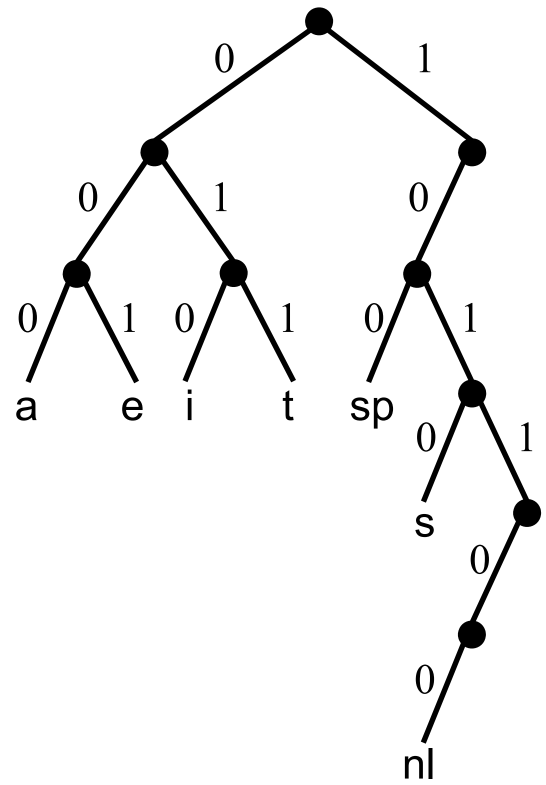


§4.2 Entropy coding

Example (Entropy 2,6644...)

Symbol	Fre-quency	Self-infor-mation	Code word length	Code
a	10	2,7225	3	000
e	15	2,1375	3	001
i	11	2,5850	3	010
s	7	3,2370	4	1010
t	9	2,8745	3	011
sp	12	2,4594	3	100
nl	2	5,0444	6	101100

Code tree



- Average code word length: 3,1969...
- Why use for „nl" six instead of four bits?

➡ **Remark:** Shannon codes are not optimal!

Properties of optimal prefix-free codes (1)

1. For an optimal code, the code tree does not have unused leaves.

§4.2 Entropy coding

Shannon-Fano codes

- **Idea:** Improve the balance of the code tree to avoid unused leaves.

```
(1) Sort the symbols by their frequency.
(2) Subdivide the symbols in this sequence into a left and
    right group, such that the sum of frequencies are as
    close as possible in both groups.
    - The two groups correspond to a left and a right sub-
      tree in the code tree.
(3) Repeat (2) for both groups until there is only one symbol
    left in the group.
```

- Application: zip-compression (`implode`-Mode)
- **Remark:** Shannon-Fano codes are not optimal.

Example

Symbol	Frequency
a	10
e	15
i	11
s	7
t	9
sp	12
nl	2

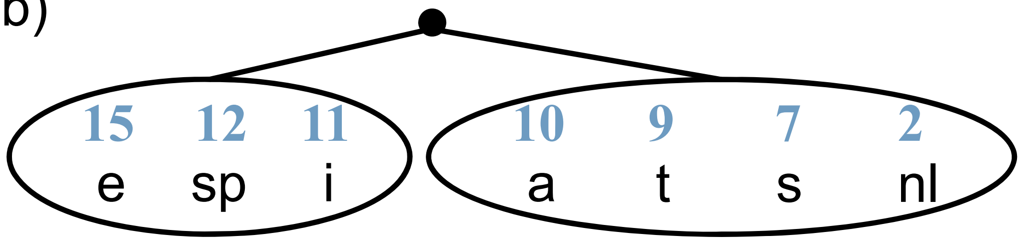
- Step**(1)**: Sort by frequency,
  symbols and frequencies as weight (blue)

  a)

15	12	11	10	9	7	2
e	sp	i	a	t	s	nl

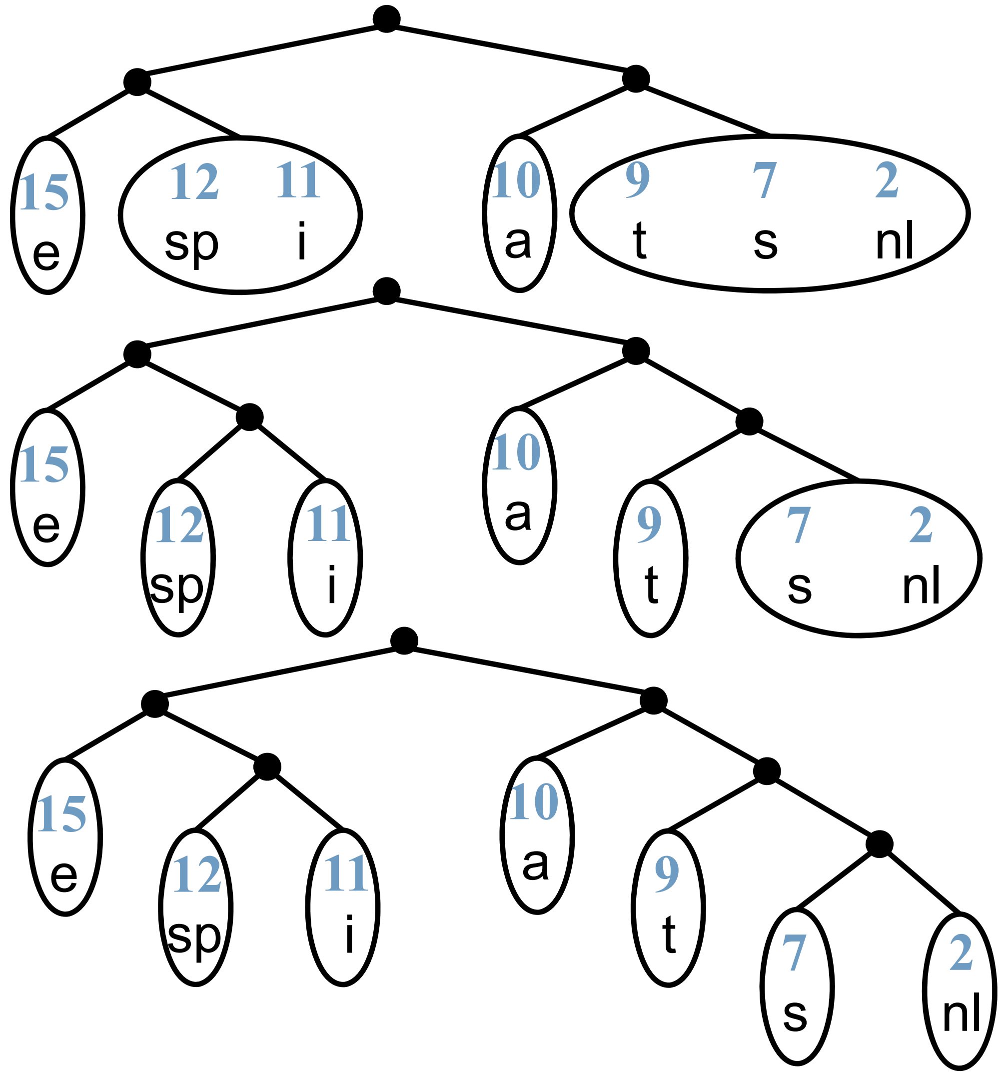
- Step**(2)** iterated:
  Subdivide into two groups of roughly equal cumulated frequency .

  b)



§4.2 Entropy coding

c)
d)
e)



Shannon-Fano code

Symbol	Frequency	Code
a	10	10
e	15	00
i	11	011
s	7	1110
t	9	110
sp	12	010
nl	2	1111

Properties of optimal prefix-free codes (1)

1. For an optimal code, the code tree does not have unused leaves.
2. There is an optimal prefix-free code, such that the code words of the two symbols u_{n-1} and u_n with least frequency differ only in the last bit.

Huffman codes (1)

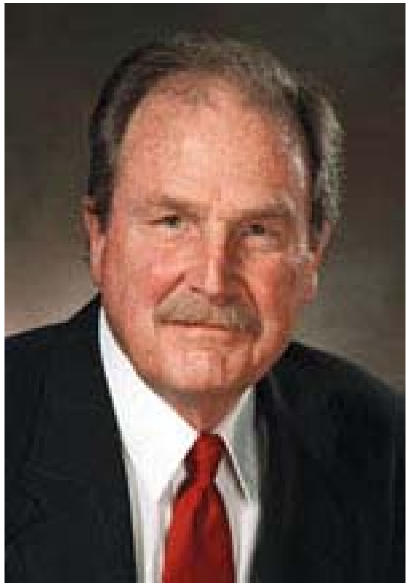
- **Idea:**

  a) Sort the symbols $u_1, \dots, u_n$ by frequency in descending order,

  b) combine the two least frequent symbols to a new symbol u'_{n-1} with cumulated frequency $p'_{n-1} = p_n + p_{n-1}$,

  c) repeat this for the new alphabet $U' = \{u_1, \dots, u_{n-2}, u'_{n-1}\}$ and

  d) stop when there is only one symbol left.



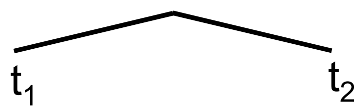
Courtesy: UC Santa Cruz

David Albert Huffman

Huffman codes (2)

(1) - Start with a forest of code trees, where each tree holds initially only one symbol.
- Each tree has a weight p, that is the sum of the frequencies of its symbols.

(2) Choose the two trees t_1 and t_2 with smallest weights p_1 and p_2 and combine them to a new tree with weight $p = p_1 + p_2$:



(3) - Repeat **(2)** until, there is only one tree left.
- This tree represents an optimal binary code for the given symbols and frequencies.

Example

- Step (1):
  Forest of trees holding exactly one symbol and frequency as weight (blue)

  a)

10	15	11	7	9	12	2
a	e	i	s	t	sp	nl

Symbol	Frequency
a	10
e	15
i	11
s	7
t	9
sp	12
nl	2

- Step (2) iterated:
  Combine the two trees with smallest weights to one common tree.

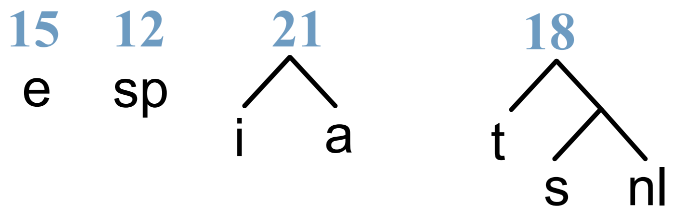
  b)

10	15	11	9	12	9
a	e	i	t	sp	s nl

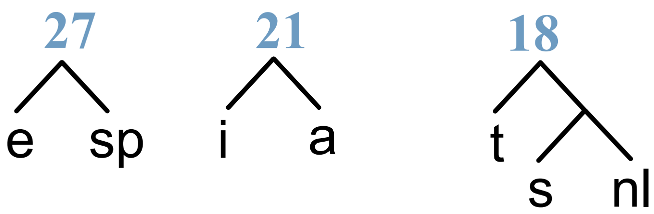
§4.2 Entropy coding

c)

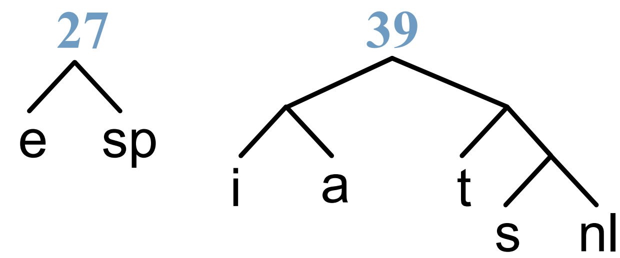
d)



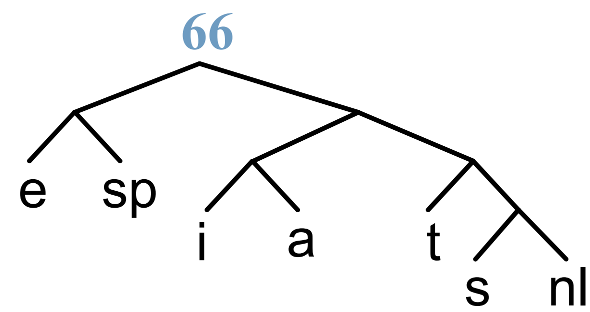
e)



f)



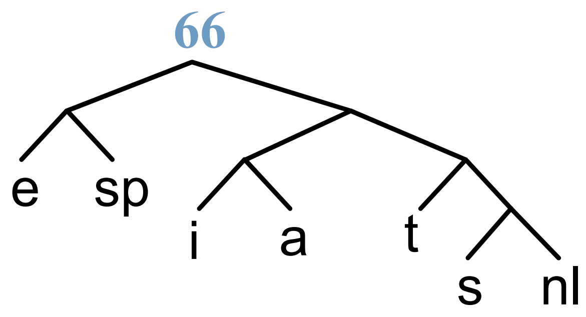
g)



h) done!

§4.2 Entropy coding

Huffman code:



Symbol	Frequency	Code
a	10	101
e	15	00
i	11	100
s	7	1110
t	9	110
sp	12	01
nl	2	1111

§4.2 Entropy coding

Properties

- Huffman codes are optimal, i.e. there is no coding that is shorter (under the given restrictions).
- The run time to compute a Huffman code for an alphabet with n symbols is $O(n \log n)$.
- **Problem:** Where do the frequencies/probabilities come from?
  - ➡ Two-Pass-Approach:
    1) Determine the frequencies of the symbols of the source.
    2) Do the Huffman coding.
  - ➡ ... or adaption, see references...

§4.2 Entropy coding

Comparison for example text with 66 symbols and entropy 2,66.

Symbol	Frequency	3-Bit-Code	Shannon Code	Shannon-Fano Code	Huffman Code
a	10	000	000	10	101
e	15	001	001	00	00
i	11	010	010	011	100
s	7	011	1010	1110	1110
t	9	100	011	110	110
sp	12	101	100	010	01
nl	2	110	101100	1111	1111
	Length of message	198	211	182	180
	Average code word length	3	3,19	2,76	2,73

Exercise

a) Construct for the source below a 3-bit-code, a Shannon code, a Shannon-Fano code and a Huffman code.

b) Compare the resulting codes with respect to their efficiency.

Symbol	a	b	c	d	e	f
Probability	0,05	0,1	0,15	0,2	0,23	0,27

§4.2 Entropy coding

Source coding theorem

- Improve the efficiency of e.g. Shannon codes:

  Combine N symbols to a vector, i.e. construct the alphabet $U \times U \times \cdots \times U = U^N =: \mathcal{U}$.

➡ The probability distribution of the new symbols is given (provided the source is memoryless) by

$$P_{\mathcal{U}} = P_{U_1} \cdot P_{U_2|U_1} \cdot P_{U_3|U_1 U_2} \ldots = P_{U_1} \cdot P_{U_2} \cdot P_{U_3} \ldots \cdot P_{U_N}$$

➡ **Source coding theorem**

The symbol sequence of a discrete memoryless source with entropy $H(U)$ can be encoded by a binary prefix-free code of tuples of N source symbols, such that the average code word length for $N \to \infty$ converges to $H(U)$ bits per symbol.

Extended Huffman Code

- The efficiency of a Huffman code can be improved by simultaneous coding of multiple symbols together, analog to the source coding theorem.

§4.2 Entropy coding

Example: Alphabet $U = \{a, b\}$, with entropy 0,469.

Symbol	P_U	Code	Symbol	P_{U^2}	Code	Symbol	P_{U^3}	Code
a	0,9	0	aa	0,81	0	aaa	0,729	0
b	0,1	1	ab	0,09	10	aab	0,081	100
			ba	0,09	110	aba	0,081	101
			bb	0,01	111	baa	0,081	110
						abb	0,009	11100
						bab	0,009	11101
						bba	0,009	11110
						bbb	0,001	11111
ACWL		1			1,290			1,598
Bits per symbol		1			0,645			0,533
Redundancy		113%			37%			13%

§4.2 Entropy coding

Problems of extended Huffman codes

- Extended Huffman codes are rarely applicable.
  - Example:
    - The probability for „a“ in the US constitution is 0,057.
    - The probability for ten „a“s is not $0{,}057^{10}$, but in reality 0.
- For an alphabet of 256 symbols and symbol sequences of length 3 the resulting alphabet has $2^{8\cdot 3}=16{,}777{,}216$ new symbols.

Content

Remarks

- Arithmetic codes belong to the class of entropy codes.
- Huffman codes are a special case of arithmetic codes.
- Arithmetic codes have almost no redundancy and can outperform any other lossless compression method.
- Today rarely used due to many patents (IBM) that limit its use.

§4.3 Arithmetic coding

- **Disadvantage of Huffman codes:** Every symbol is mapped to an integer number of bits.
  - ➡ The symbols are mapped individually depending on their probability to code words.
  - ➡ If a symbol has a probability, that is not a power of ½, the corresponding „fracture of a bit“ has to be filled to the next bit position.
  - ➡ There is redundancy for every symbol.
- **Disadvantage of extended Huffman codes:** For an alphabet of 256 symbols and symbol sequences of length 3 the resulting new alphabet consists of $2^{8 \cdot 3} = 16.777.216$ new symbols.

§4.3 Arithmetic coding

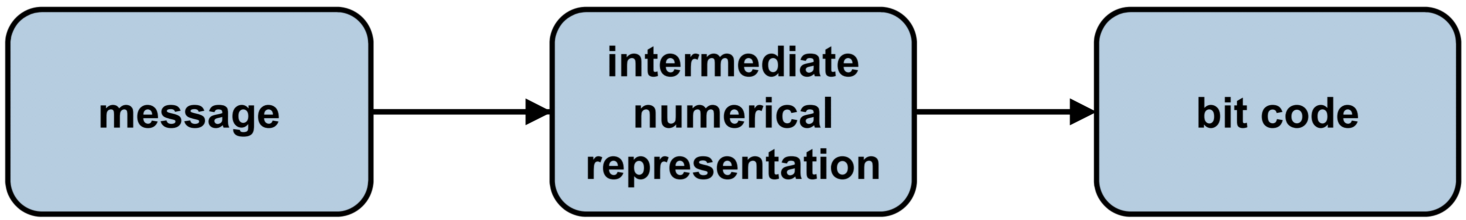
- **Idea:** Code complete words or messages using intervals.
  - ➡ Internally fractional bit positions can be used.
  - ➡ There is only redundancy for the complete message.

§4.3 Arithmetic coding

Approach for arithmetic coding

- **Given:** **Message** $a_{i_1} a_{i_2} \dots a_{i_m} \in \Sigma^m$ from the
  Alphabet $\Sigma = \{a_1, \dots, a_n\}$ equipped with
  Occurrence probabilities $P(a_i) = p_i$ for the symbols.

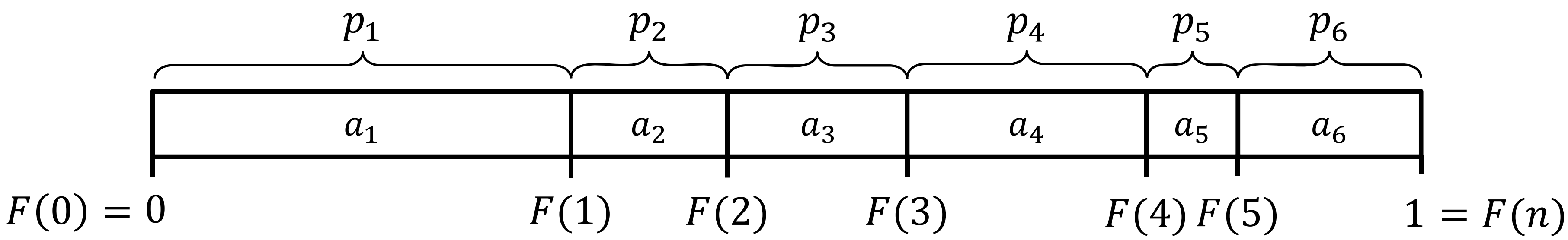
1. Compute first an **intermediate numerical representation** $\in [0,1)$ for the message.
2. Compute **binary code** for this intermediate numerical representation.



§4.3 Arithmetic coding

Compute the intermediate numerical representation (1)

- Every symbol has an occurrence probability p_i.
- ➡ These occurrence probabilities sum to one, i.e. $\sum p_i = 1$.
- ➡ The occurrence probabilities partition the interval $[0,1]$.
- ➡ Using the **cumulated probabilities** $F(i) = \sum_{k=1}^{i} p_k \in [0,1]$ the following partition is defined



- ➡ To represent the symbol a_{i_k} of the message, choose the interval that contains a_{i_k}.

§4.3 Arithmetic coding

Compute the intermediate numerical representation (2)

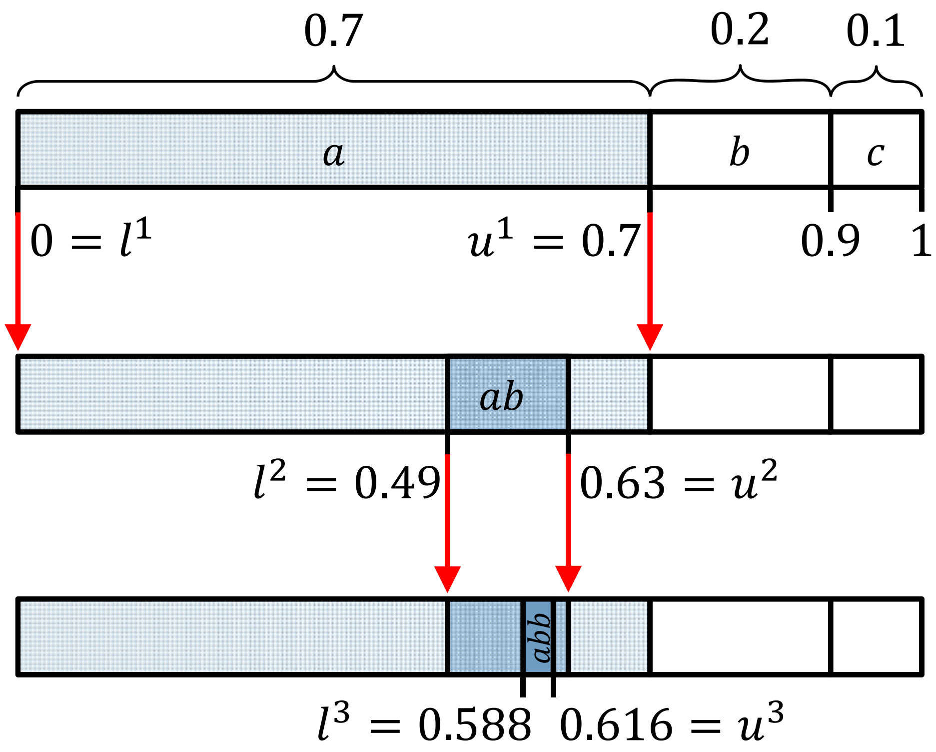
- Iterate this approach:
  - The $(k-1)$-th interval is partitioned according to the occurrence probabilities.
  - The smaller the interval, the less likely is the symbol sequence and correspondingly the information content is high.
  - ➡ Many bits are necessary to code this small interval (entropy coding).
- The corresponding intervals can be determined recursively:

$\boldsymbol{k=1}$: $I^1 = [l^1, u^1)$ with $l^1 = F(i_1 - 1)$ and $u^1 = F(i_1)$.

$\boldsymbol{k=2,\dots,m}$: $I^k = [l^k, u^k)$ with $l^k = l^{k-1} + (u^{k-1} - l^{k-1})F(i_k - 1)$
and $u^k = l^{k-1} + (u^{k-1} - l^{k-1})F(i_k)$

Compute the intermediate numerical representation (3)

- **Example:** Message $abb \in \Sigma^3$ from the alphabet $\Sigma = \{a, b, c\}$ with occurrence probabilities for the symbols
  $P(a) = 0.7,$
  $P(b) = 0.2,$
  $P(c) = 0.1.$



➡ The interval $[0.588;\ 0.616)$ codes every message, that begins with abb.

§4.3 Arithmetic coding

Compute the intermediate numerical representation (4)

- **Common numerical representation** by interval midpoint

$$T\left(a_{i_1} a_{i_2} \dots a_{i_m}\right) = \frac{l^m + u^m}{2}.$$

➡ The only information required to de-code $T\left(a_{i_1} a_{i_2} \dots a_{i_m}\right)$ is the cumulated probability distribution F.

```
Initialize l^0 = 0 and u^0 = 1;
for k=1,...,m do {
   Find i_k such that F(i_k − 1) ≤ T < F(i_k);
   output(a_{i_k});
   Update l^k and u^k;
}
```

§4.3 Arithmetic coding

Binary representation of the numerical representation

Self-information of the message

$$\ell(a_{i_1}a_{i_2}\ldots a_{i_m}) = \left\lceil \log_2 \frac{1}{p_{i_1}\cdot p_{i_2}\cdot\ldots\cdot p_{i_m}} \right\rceil + 1.$$

- **Binary code:** Use as code $C(a_{i_1}a_{i_2}\ldots a_{i_m})$ the first $\ell(a_{i_1}a_{i_2}\ldots a_{i_m})$ leading bits of the numerical representation $T(a_{i_1}a_{i_2}\ldots a_{i_m})$.
- **Example:** $\Sigma = \{a, b\}$, $P(a) = 0.9$, $P(b) = 0.1$, $m = 2$.

word x	$T(x)$	binary T	$\ell(x)$	code C
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

Properties

- How effective is arithmetic coding?

  The arithmetic codes of two different messages are prefix-free.

- How efficient is arithmetic coding?

  For the average code word length $L(\Sigma)$ per symbol of an arithmetic code with parameter m and an alphabet with entropy $H(\Sigma)$ the following inequality holds:

  $$H(\Sigma) \leq L(\Sigma) \leq H(\Sigma) + 2/m.$$

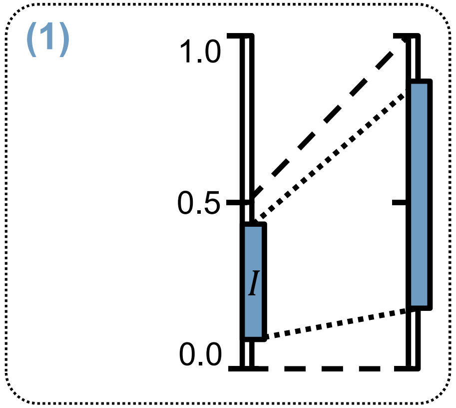
  (compare to Shannon-Codes)

- **Problem:** Accuracy of the computation, because the length of the intervals converges to zero.

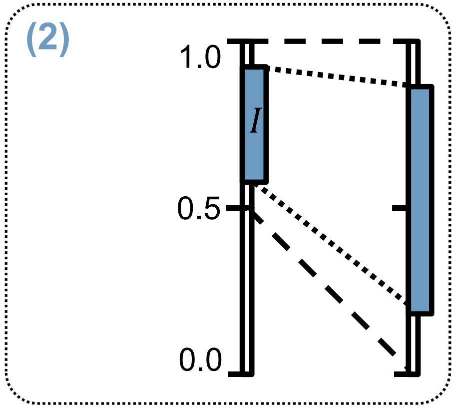
Re-scaling (1)

- **Solution:** Re-scaling of the intervals by factor 2:

(1) $I \subseteq [0,0.5)$: Multiply l^k, u^k by 2 and output bit „0“.



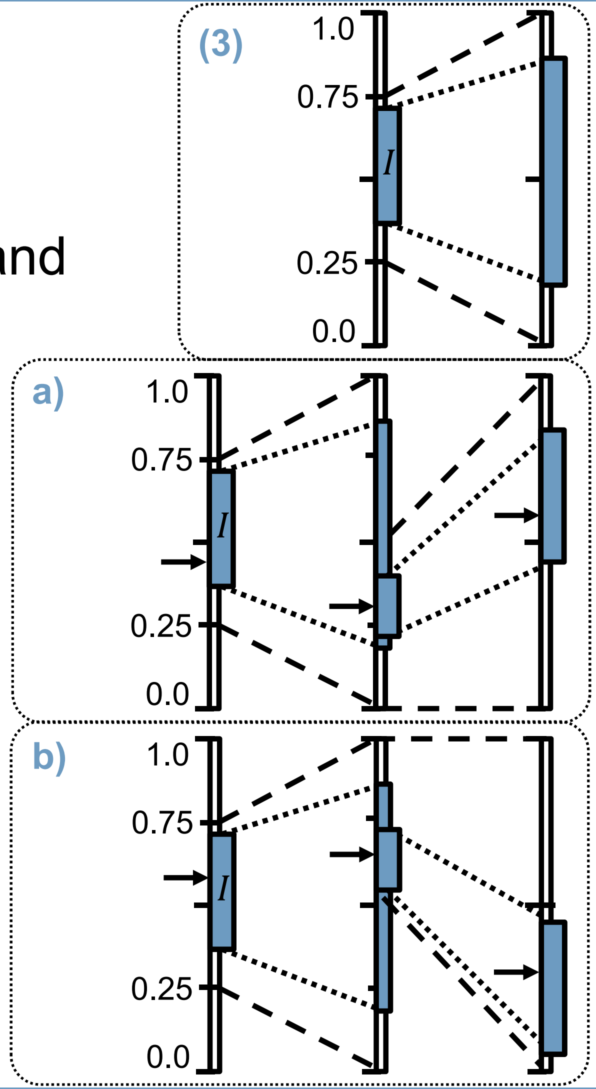
(2) $I \subseteq [0.5,1)$: Translate l^k, u^k by -0.5, multiply by 2 and output bit „1“.



§4.3 Arithmetic coding

Re-scaling (2)

(3) $I \subseteq [0.25, 0.75)$: Translate l^k, u^k by −0.25, multiply by 2 and withhold output of any bit.



a) After re-scaling case (1) occurs, then re-scale as in case (1) and output bits „01“.

b) After re-scaling case(2) occurs, then re-scale as in case (2) and output bits „10“.

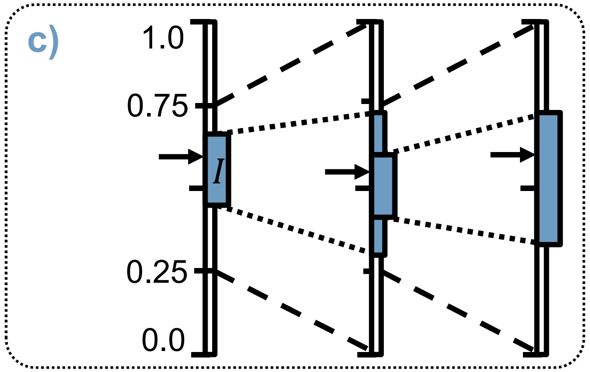
Re-scaling (3)

(3) $I \subseteq [0.25, 0.75)$: ...

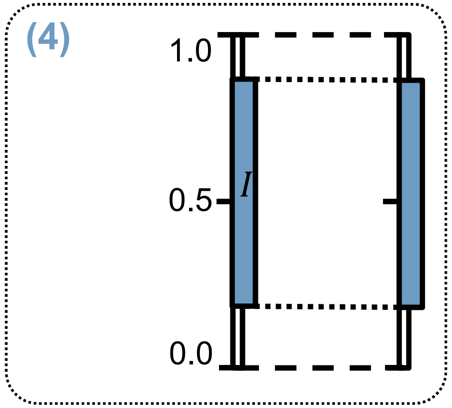
a) ...

b) ...

c) After re-scaling case (3) occurs, then re-scale as in case (3), withhold output of any bit and iterate...



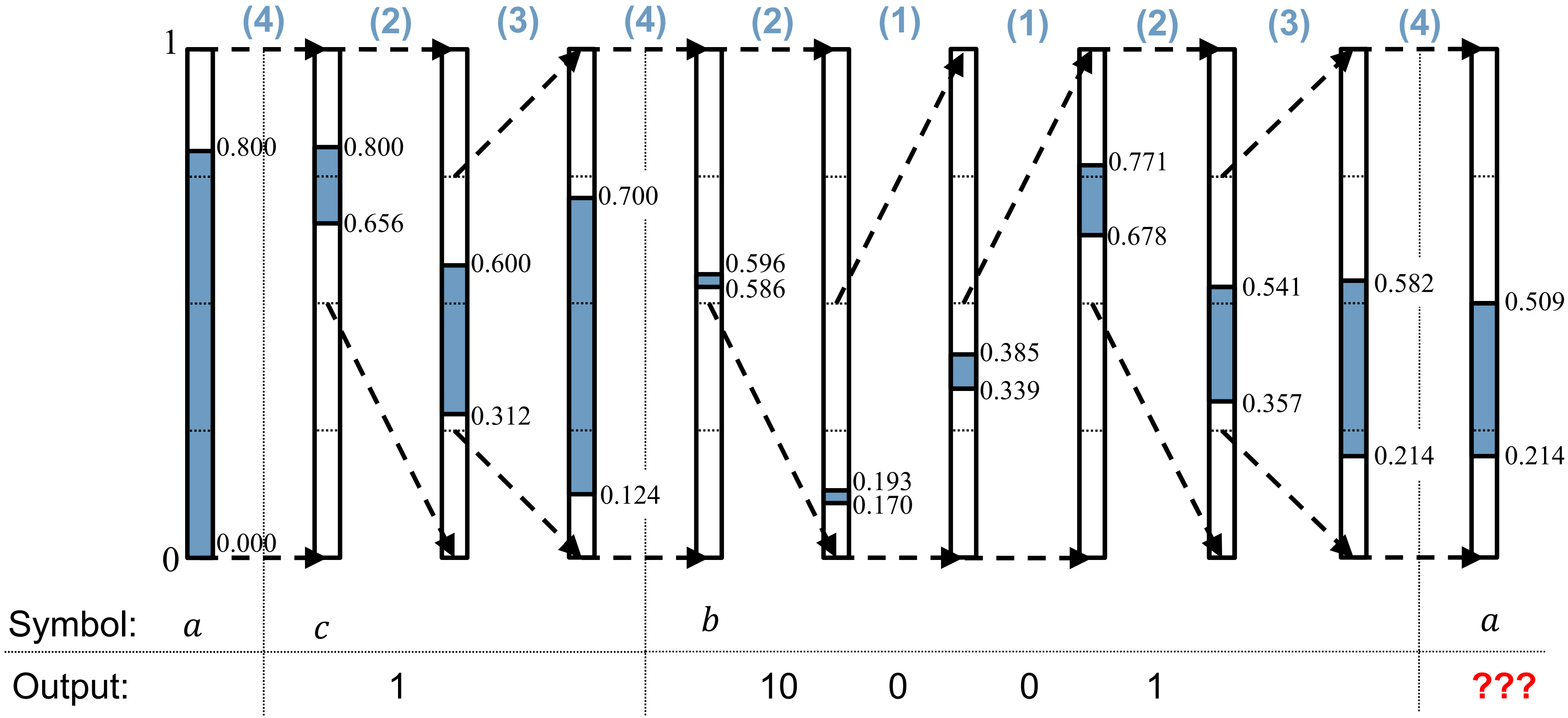
(4) otherwise: no re-scaling and withhold output of any bit.



§4.3 Arithmetic coding

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

- Encode message $acba$.



Arithmetic coding with re-scaling (1)

```
// btf = bits to follow
```

Input: Message $a_{i_1} a_{i_2} \dots a_{i_m} \in \Sigma^m$;
$l = 0$; $u = 1$; btf=0; // Initialization
while (There are still un-coded symbols) **do** {
  Read a_{i_k}; $d = u - l$; $u = l + d \cdot F(i_k)$; $l = l + d \cdot F(i_k - 1)$;
  while (1) { // Cases (1)-(4)
    if ($u \leq 0.5$) { // Case (1)
      output(0); **while** (btf>0){ output(1); btf--; }
      $l = 2l$; $u = 2u$; // Re-scale $[0, 0.5) \to [0, 1)$
    }
    else if ($l \geq 0.5$) { // Case (2)
      output(1); **while** (btf>0){ output(0); btf--; }
      $l = 2l - 1$; $u = 2u - 1$; // Re-scale $[0.5, 1) \to [0, 1)$
    }
    else ...
  }
  ...

Arithmetic coding with re-scaling (2)

```
...                                          // Initialization
while (There are still un-coded symbols) do {
   ...
   while (1) {                               // Cases (1)-(4)
      ...
      else if ([l,u) ⊆ [0.25,0.75)) {        // Case (3)
         btf++;
         l = 2l − 0.5;  u = 2u − 0.5;        // Re-scale [0.25,0.75) → [0,1)
      }
      else break;                            // Case (4)
   }
   ...
}
```

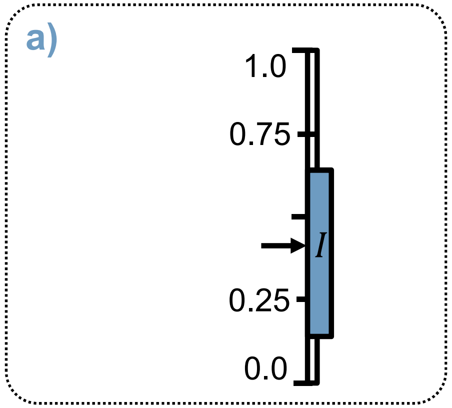
§4.3 Arithmetic coding

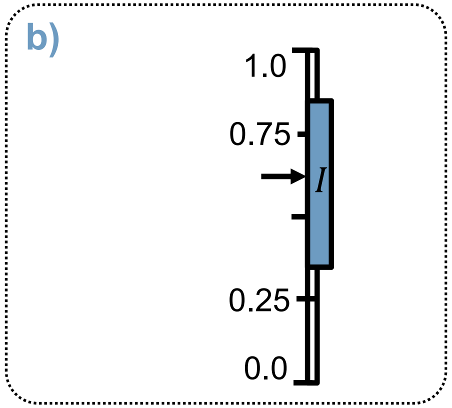
Re-scaling (4)

(5) **Termination of coding:** There are no un-coded symbols left, but there are bits left for output.

a) $\frac{l^m+u^m}{2} < 0.5$: output bits „01…1“.

b) $\frac{l^m+u^m}{2} \geq 0.5$: output bits „10…0“.





Arithmetic coding with re-scaling (3)

```
...                                         // Initialization
while (There are still un-coded symbols) do {
   ...
   while (1) {...}                          // Cases (1)-(4)
   btf++;                                   // End of coding
   T = (l + u)/2;
   if (T < 0.5) {
      output(0); while (btf>0) { output(1); btf--; }
   }
   else {
      output(1); while (btf>0) { output(0); btf--; }
   }
}
```

§4.3 Arithmetic coding

Example

symbol	interval			case	btf	output
	l	u	d			
a	0.0	0.8	0.8	(4)	0	
c	0.656	0.8	0.144	(2)	0	1
	0.312	0.6	0.288	(3)	1	
	0.124	0.7	0.576	(4)	1	
b	0.5848	0.59632	0.01152	(2)	0	10
	0.1696	0.19264	0.02304	(1)	0	0
	0.3392	0.38528	0.04608	(1)	0	0
	0.6784	0.77056	0.09216	(2)	0	1
	0.3568	0.54112	0.18432	(3)	1	
	0.2136	0.58224	0.36864	(4)	1	
a	0.2136	0.508512	0.294912	(4)	1	
	0.2136	0.508512	0.294912	(5)	2	011

§4.3 Arithmetic coding

```
Input: Message a_{i_1} a_{i_2} ... a_{i_m} ∈ Σ^m;
l = 0;  u = 1;  btf=0;                                  // Initialization
while (There are still un-coded symbols) do {
    Read a_{i_k};  d = u − l;  u = l + d · F(i_k);  l = l + d · F(i_k − 1);
    while (1) {                                         // Cases (1)-(4)
        if (u ≤ 0.5) {                                  // Case (1)
            output(0); while (btf>0){ output(1); btf--; }
            l = 2l;  u = 2u;                            // Re-scale [0,0.5) → [0,1)
        }
        else if (l ≥ 0.5) {                             // Case (2)
            output(1); while (btf>0){ output(0); btf--; }
            l = 2l − 1;  u = 2u − 1;                    // Re-scale [0.5,1) → [0,1)
        }
        else if ([l,u) ⊆ [0.25,0.75)) {                 // Case (3)
            btf++;
            l = 2l − 0.5;  u = 2u − 0.5;                // Re-scale [0.25,0.75) → [0,1)
        }
        else break;                                     // Case (4)
    }
    btf++;                                              // End of coding
    T = (l + u)/2;
    if (T < 0.5) {
        output(0); while (btf>0) { output(1); btf--; }
    }
    else {
        output(1); while (btf>0) { output(0); btf--; }
    }
}
```

§4.3 Arithmetic coding

De-coding (1)

- The de-coder works exactly like the coder.
- Additionally the parameter m must be transmitted first (i.e. the length of the coded symbol sequences) or the method defines a constant m a-priori.
  - The de-coder has to output exactly m symbols (or less, if the termination symbol is known).
- Problem of termination.
  - Solution 1: The length of the last sequence of symbols has to be transmitted separately, or
  - Solution 2: A special termination symbol is added to the alphabet endowed with a low probability.

De-coding(2)

- The de-coder needs a parameter for the accuracy of the representation (i.e. number of digits h in the binary representation) of the interval bounds for the tests, to detect the interval containing the code word.
- How to choose this parameter?

➡ The number of digits h must be large enough to separate the intervals sufficiently from each other, i.e.

$$\min_i p_i > 2^{-h+2}.$$

§4.3 Arithmetic coding

De-coding(3)

```
Input: Code word c_1 c_2 ... c_n ∈ {0,1}^n and number of digits h;
l = 0;  u = 1;  r = 1;                    // Initialization
for (k = 1, ..., m) {
   d = u − l;
   Find i_k such that l + d · F(i_k − 1) ≤ 0.c_r c_{r+1} ... c_{r+h} < u + d · F(i_k);
   u = l + d · F(i_k);  l = l + d · F(i_k − 1);
   output(a_{i_k});
   while (1) {                            // Cases (1)-(4)
      if (u ≤ 0.5) {                      // Case (1)
         r++;  l = 2l;  u = 2u;           // Re-scale [0, 0.5) → [0, 1)
      }
      else if (l ≥ 0.5) {                 // Case (2)
         r++;  l = 2l − 1;  u = 2u − 1;   // Re-scale [0.5, 1) → [0, 1)
      }
      else ...
 }}
```

De-coding(4)

```
...                                        // Initialization
for (k = 1, ..., m) {
   ...
   while (1) {                             // Cases (1)-(4)
      ...
      else if ([l,u) ⊆ [0.25,0.75)) {      // Case (3)
         r++;  c_r = 1 - c_r;
         l = 2l - 0.5;  u = 2u - 0.5;      // Re-scale [0.25,0.75) → [0,1)
      }
      else break;                          // Case (4)
   }
}
```

§4.3 Arithmetic coding

```
Input: Code word c_1 c_2 ... c_n ∈ {0,1}^n and number of digits h;
l = 0;  u = 1;  r = 1;                              // Initialization
for (k = 1, ..., m) {
    d = u − l;
    Find i_k such that l + d · F(i_k − 1) ≤ 0.c_r c_{r+1} ... c_{r+h} < u + d · F(i_k);
    u = l + d · F(i_k);  l = l + d · F(i_k − 1);
    output(a_{i_k});
    while (1) {                                     // Cases (1)-(4)
        if (u ≤ 0.5) {                              // Case (1)
            r++;  l = 2l;  u = 2u;                  // Re-scale [0,0.5) → [0,1)
        }
        else if (l ≥ 0.5) {                         // Case (2)
            r++;  l = 2l − 1;  u = 2u − 1;          // Re-scale [0.5,1) → [0,1)
        }
        else if ([l,u) ⊆ [0.25,0.75)) {             // Case (3)
            r++;  c_r = 1 − c_r;
            l = 2l − 0.5;  u = 2u − 0.5;            // Re-scale [0.25,0.75) → [0,1)
        }
        else break;                                 // Case (4)
    }
}
```

Content

§4.4 LZW coding

- For Shannon and Huffman codes the statistic of the source needs to be known a-priori for the construction of the code.
  - If the statistic is unknown, the code itself needs to be transmitted.

➡ **Universal source coding:** Methods that work without a-priori information about the source statistic and learn the source statistic during input of the message.

§4.4 LZW coding

Dictionary-based compression methods

- Besides the different frequencies of symbols in the text, also use frequencies of symbol sub-sequences (e.g. words) in the text.
- **Dictionary:** Coding of words instead of coding of individual symbols.

➡ **Source with memory.**

- **Static:** Dictionary is defined before coding starts and will not be changed in the process.
- **Dynamic:** Dictionary adapts itself to the actual text, that needs to be compressed, dynamically.

§4.4 LZW coding

Lempel-Ziv‘s Idea

- Construct the dictionary simultaneously with the coding of the text.

Welch‘s Idea

- Initialize the dictionary with the symbols of the alphabet.

§4.4 LZW coding

- **Remark:** There are many variants of the Lempel-Ziv method, which are known as LZ77, LZ78, LZW, LZSS, ROLZ, LZMA, etc.
  - We will discuss here only LZW as a representative member for this class of compression methods.
- **Examples:**
  - **Lempel-Ziv:** zip, TIFF (Image File Format)
  - **Lempel-Ziv-Welch:** Compress in Unix
  - **Lempel-Ziv-Markov:** 7-zip

§4.4 LZW coding

Example („Monster in the mirror“, Sesame street)

Text: `wabba wabba wabba wabba woo woo woo`

1. `wabba wabba wabba wabba woo woo woo`
2. `wabba wabba wabba wabba woo woo woo`
3. `wabba wabba wabba wabba woo woo woo`
4. `wabba wabba wabba wabba woo woo woo`
5. `wabba wabba wabba wabba woo woo woo`
6. `wabba wabba wabba wabba woo woo woo`
7. `wabba wabba wabba wabba woo woo woo`
8. . . .

Dictionary:

index	entry	output
1	_	
2	a	
3	b	
4	o	
5	w	
6	wa	5
7	ab	5 2
8	bb	5 2 3
9	ba	5 2 3 3
10	a_	5 2 3 3 2
11	_w	5 2 3 3 2 1
12	wab	5 2 3 3 2 1 6

§4.4 LZW coding

Output:

5 2 3 3 2 1 6 8 10 12 9
11 7 16 5 4 4 11 21 23 4

Final dictionary:

index	entry	index	entry
1	_	14	a_w
2	a	15	wabb
3	b	16	ba_
4	o	17	_wa
5	w	18	abb
6	wa	19	ba_w
7	ab	20	wa
8	bb	21	oo
9	ba	22	o_
10	a_	23	_wo
11	_w	24	oo_
12	wab	25	_woo
13	bba		

LZW coding

```
Input:  Text T=t1...tn over the alphabet U
Output: Coding of T

Initialize dictionary D using the symbols of U;
Initialize string s=t1;
while (There are still un-read symbols in T) do
   read next symbol c of T;
   if (s+c is beginning of an entry in D)
   then s=s+c;      // determine the actual match
   else {  return code word for s;
           add s+c to D;
           s=c;  }
end while;
Return code word for s;
```

§4.4 LZW coding

Example („Monster in the mirror“, Sesame street)

Input: 5 2 3 3 2 1 6 8 10 12 9 11 7...

1. 5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4...
2. 5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4...
3. 5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4...
4. 5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4...
5. 5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4...
6. 5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4...
7. 5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4...
8. ...

Dictionary:

index	entry	output
1	_	
2	a	
3	b	
4	o	
5	w	
6	wa	wa
7	ab	wab
8	bb	wabb
9	ba	wabba
10	a_	wabba_
11	_w	wabba_wa
12	wab	wabba_wabb

§4.4 LZW coding

LZW de-coding

```
Input:  Sequence of code words c1...cm
Output: Sequence of symbols over the alphabet U

Initialize dictionary D using the symbols of U;
Read i=c1 and return symbol D[i], that is coded by i;
while (There are still un-read code words) do
   Read next code word j;
   if (j is in D)
   then { Add    D[i] + firstchar(D[j]) to D;
          return D[j]                       ; }
   else { Add    D[i] + firstchar(D[i]) to D;
          return D[i] + firstchar(D[i])     ; }
   i=j;
end while;
```

§4.4 LZW coding

Exercise

a) De-code the LZW code `1 2 4 3 5 8` using the following dictionary.

Index	Entry
1	a
2	b
3	c

b) Test your result, by encoding the de-coded text using LZW coding.

§4.4 LZW coding

Properties

- Dictionary adapts dynamically to the target text.
  - ➡ It will contain finally the most frequently occurring symbol sequences.
- Dictionary needs to be transmitted, too.
  - Only the initial dictionary of the alphabet needs to be known, all the rest is generated dynamically in the process of en-/de-coding.
- Coding and de-coding can be done in linear run-time.
- LZW codes yield in general better compression rates than Huffman codes.

Goals

- What is lossless compression? Which methods do you know?
- How do RLE or LZW work?
- What is entropy coding?
- How do you compute a Huffman code and what is the run-time for this algorithm?
- How do Huffman coding or arithmetic coding work?