Introduction to Compression Algorithms

Jindřich Nový

Software Engineer

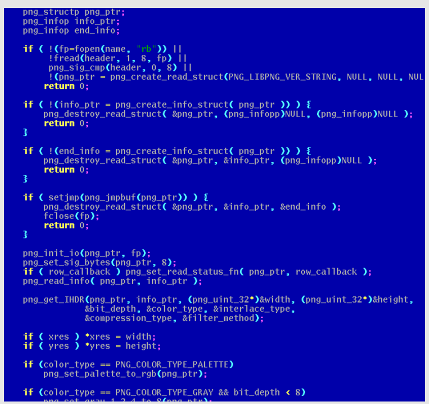
jnovy@redhat.com

Why compression?

Stored information is redundant.

57% of the original size losslessly (PNG)

11% with lossy compression (JPEG)



16% losslessly (bzip2)

Sequential Coders

RLE Encoding 1

Substitutes a sequence of the same letters with double-code which contains the letter itself and sequence length.

“abaaaacabbcaaaa” 15 B – input message

4 2 4

“a1b1a4c1a1b2c1a4” 16 B – [letter,length] - output

compression ratio: 16/15 = 1.06
minimal compression ratio: 2/15 = 0.13
maximal compression ratio: 30/15 = 2.00

RLE Encoding 2

Do not encode unique letters.

"abaaaacabbcaaaa" 15 B – input message

4 2 4

"aba4cab2ca4" 11 B – not reconstructable

"abaa4cabb2caa4" 14 B – [letter] or [letter,letter,length]

compression ratio: 14/15 = 0.93

minimal compression ratio: 3/15 = 0.20

maximal compression ratio: 22/15 = 0.68

The Algorithm – Encoder

I="abaaaacabbcaaaa" 15 B – input message

O="abaa4cabb2caa4" 14 B – output message

```
X=get letter from I
Y=get letter from I
if X==Y
    write X,Y to O
    N=count repeated letters in I
    write N+2 to O
    go to
else
    write X to O
    X=Y
    Y=get letter from I
    go to
```

The Algorithm – Decoder

I="abaa4cabb2caa4" 14 B – input message

O="abaaaacabbcaaaa" 15 B – output message

```
X=get letter from I
write X to O
Y=get letter from I
if X==Y
    N=get letter from I
    write N-1 letters X to O
    go to
else
    write Y to O
    X=Y
    go to
```

RLE Encoding 3

Enhancement: “skipping RLE”

“abaaaacabbcaaaa” 15 B – input message

4 2 4

“2ab4a5cabbc4a” 13 B – [length]

compression ratio: 13/15 = 0.86

minimal compression ratio: 3/15 = 0.20

maximal compression ratio: 16/15 = 1.06

RLE Algorithms Conclusion

... used mostly as a first pass metod
... it's fast, $O(n)$
... ineffective

Burrows-Wheeler Transform

(block-sorting compression)

BWT - Coder

1. generate rotated sequence of the input message
2. sort sequences lexicographycally
3. output is the last letters from the message + starting index, where I_f=1

I_f	rotated M
0	abaca
1	bacaa
2	acaab
3	caaba
4	aabac

I	I_f	rotated, sorted M	output M_{BWT}
0	4	aabac	c
1	0	abaca	a
2	2	acaab	b
3	1	bacaa	a
4	3	caaba	a

cabaa, 3

BWT - Decoder

The input to the decoder is:

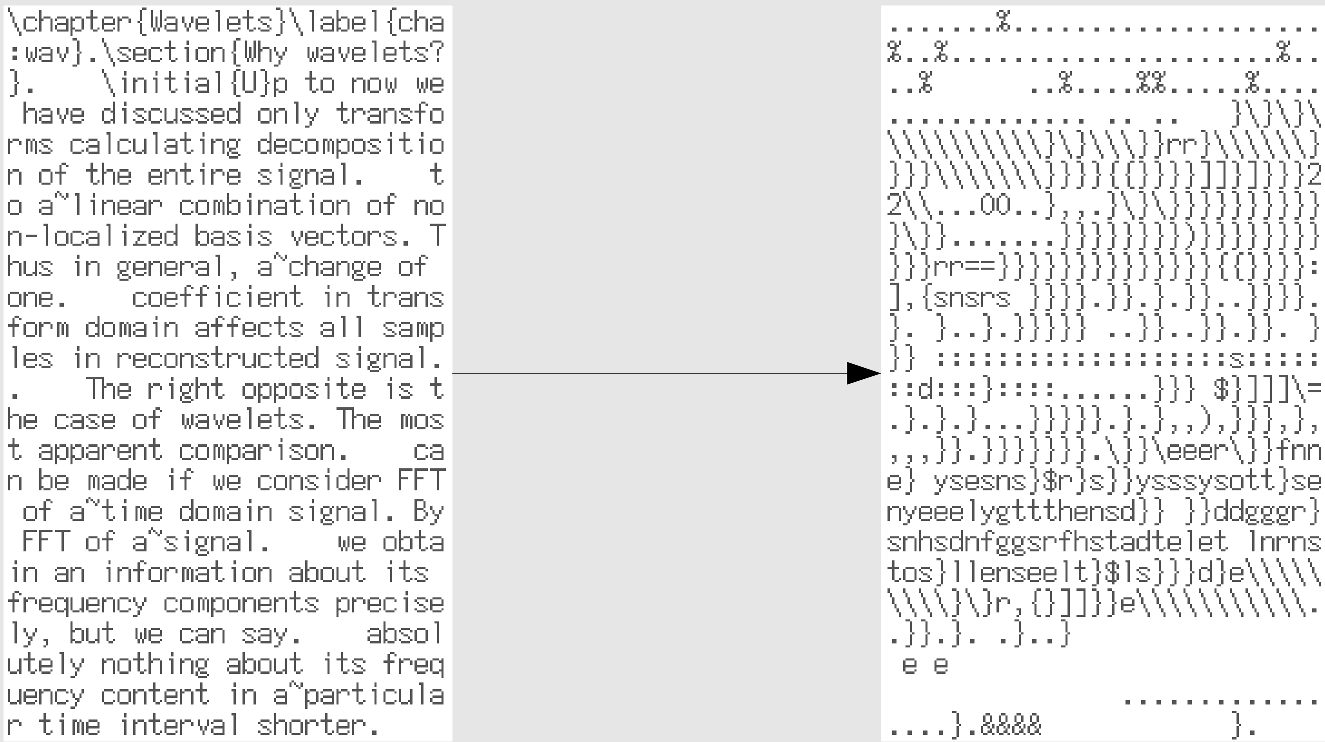
`cabaa,3`

I		M_{BWT}		I_b	sorted M_{BWT}
0	$\stackrel{4}{\Longrightarrow}$	c	$\stackrel{4}{\Longrightarrow}$	1	a
1	$\stackrel{5}{\Longrightarrow}$	a	$\stackrel{5}{\Longrightarrow}$	3	a
2	$\stackrel{2}{\Longrightarrow}$	b	$\stackrel{2}{\Longrightarrow}$	4	a
3	$\stackrel{1}{\Longrightarrow}$	a	$\stackrel{1}{\Longrightarrow}$	2	b
4	$\stackrel{3}{\Longrightarrow}$	a	$\stackrel{3}{\Longrightarrow}$	0	c

1. $i = I_1$
2. $M_{\mathrm{BWT}}(i) \rightarrow$ output
3. $i = I_b(i)$
4. if $i \neq I_0$ go to 2

BWT – Why is it used?

Burrows-Wheeler Transform doesn't compress the input message, so why would we bother using it?



Lempel-Ziv Algorithms

The first dictionary based compression method was created by Abraham Lempel and Jacob Ziv – Lempel-Ziv-77 - LZ77.

The output message contains symbols of three types:

1. letter = uncompressed letter
2. code word = contains [position,length]
3. flag = a bit telling encoder to expect either a letter or a code word

Lempel-Ziv 77 Algorithm

"abaaaacabbcaaaa"

```
 0123456789ABCDE
"abaaaacabbcaaaa"

 0000101011
"abaa2c0b52"
     2 2 23
```

Assuming:
- 4 bits for position (0..15)
- 4 bits for length (1..16)
- 8 bits for a letter (0..255)

... and the compression ratio is:

$(6 \cdot 8+10+8 \cdot 4)/14 \cdot 8 = 0.80$

Lempel-Ziv 77 – Finite Window

Finite window encoding principle:

- position of a match is set relatively from encoding position
- size of a window where a match is searched is limited by bits to express the match position

Let's see: 2-bit position:

```
 3210
“abaaaacabbcaaaa”
```

or later in encoding process:

```
    3210
“abaaaacabbcaaaa”
```

Lempel-Ziv 77 – Finite Window

input: “abaaaacabbcaaaa”

output: 0
“a”

coding position: 0
bits per position: 3 (0..7)
bits per length: 1 (1..2)
bits per letter: 8 (0..255)

Lempel-Ziv 77 – Finite Window

input: 0
"abaaaacabbcaaaa"

output: 00
"ab"

coding position: 1
bits per position: 3 (0..7)
bits per length: 1 (1..2)
bits per letter: 8 (0..255)

Lempel-Ziv 77 – Finite Window

10
input: “abaaaacabbcaaaa”

output: 001
“ab1”
1

coding position: 2
bits per position: 3 (0..7)
bits per length: 1 (1..2)
bits per letter: 8 (0..255)

Lempel-Ziv 77 – Finite Window

	210
input:	“abaaaacabbcaaaa”
output:	0011
	“ab10”
	11

coding position:	3	
bits per position:	3	(0..7)
bits per length:	1	(1..2)
bits per letter:	8	(0..255)

Lempel-Ziv 77 – Finite Window

```
                3210
input:         "abaaaacabbcaaaa"

output:         00111
               "ab101"
                  112
```

coding position: 4
bits per position: 3 (0..7)
bits per length: 1 (1..2)
bits per letter: 8 (0..255)

Lempel-Ziv 77 – Finite Window

```
              543210
input:       “abaaaacabbcaaaa”

output:       001110
             “ab101c”
                112
```

coding position: 6
bits per position: 3 (0..7)
bits per length: 1 (1..2)
bits per letter: 8 (0..255)

Lempel-Ziv 77 – Finite Window

```
        6543210
input:  “abaaaacabbcaaaa”

output:  0011101
        “ab101c6”
           112 2
```

coding position: 7

bits per position: 3 (0..7)

bits per length: 1 (1..2)

bits per letter: 8 (0..255)

Lempel-Ziv 77 – Finite Window

```
              76543210
input:      “abaaaacabbcaaaa”

output:      00111011
            “ab101c60”
               112 21
```

coding position:	9	
bits per position:	3	(0..7)
bits per length:	1	(1..2)
bits per letter:	8	(0..255)

Lempel-Ziv 77 – Finite Window

```
          76543210
input:    "abaaaacabbcaaaa"

output:    001110111
          "ab101c603"
             112 212
```

coding position: 10

bits per position: 3 (0..7)

bits per length: 1 (1..2)

bits per letter: 8 (0..255)

Lempel-Ziv 77 – Finite Window

```
               76543210
input:     “abaaaacabbcaaaa”

output:     0011101111
           “ab101c6037”
              112 2122
```

coding position:	12	
bits per position:	3	(0..7)
bits per length:	1	(1..2)
bits per letter:	8	(0..255)

Lempel-Ziv 77 – Finite Window

```
                76543210
input:   “abaaaacabbcaaaa”

output:   00111011111
         “ab101c60370”
            112 21221
```

coding position: 14
bits per position: 3 (0..7)
bits per length: 1 (1..2)
bits per letter: 8 (0..255)

Lempel-Ziv 77 – Results

output: 00111011111
“ab101c60370”
112 21221

output in binary:

0 01100001 0 01100010 1 001 0 1 000 0 1 001 1
0 01100011 1 110 1 1 000 0 1 011 1 1 111 1
1 000 0

Bits of the original message: $15 \cdot 8 = 120$

Bits of the compressed message: $3 \cdot 8 + 8 \cdot (3+1) + 12 \cdot 1 = 64$

Compression ratio: 0.53

We losslessly compressed the message to 53% of its length.

Lempel-Ziv-Welch Algorithm

Terry Welch invented a modification of LZ77 in 1984 called LZW. It is the first fully dictionary based method.

Extends letter alphabet of a certain number of bits.

For instance:
each 8-bit letter (0..255) within a message extends to 9 bits (0..511), where:

(0..255)	are considered an original letter,
(256..511)	are considered links to a created dictionary.

Statistical Coders

(aka Entropy Coders)

What's Entropy?

Let's have an alphabet $L=\{l_0, l_1, \dots, l_{N-1}\}$, of N letters.

When the probability distribution $P=\{p_0, p_1, \dots, p_{N-1}\}$ is known, then the amount of information needed to encode a letter is $H(l_n)$, the entropy of the alphabet is $H(L)$:

$$H(l_n) = -\log_2 p_n, \qquad H(L) = \sum_{n=0}^{N-1} p_n H(l_n),$$

$$H(L) = -\sum_{n=0}^{N-1} p_n \log_2 p_n$$

Shannon-Fano Coder

input message: “`2ab4a5cabbc4a`”

order	letter	count	p(letter)	H(letter)
1	a	4	0.30	1.70
2	b	3	0.23	2.11
3	c	2	0.15	2.70
4	4	2	0.15	2.70
5	2	1	0.07	3.70
6	5	1	0.07	3.70

Shannon-Fano Coder

"2ab4a5cabbc4a"

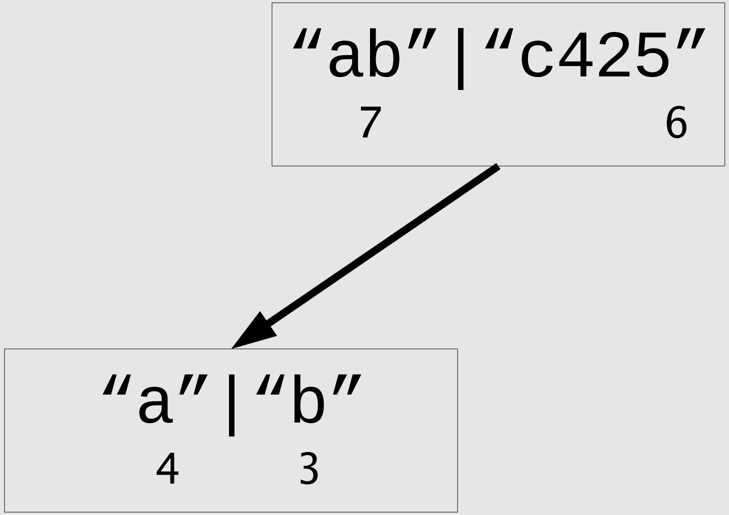
"ab" | "c425"
7 6

letter	count
a	4
b	3
c	2
4	2
2	1
5	1

Shannon-Fano Coder

“2ab4a5cabbc4a”

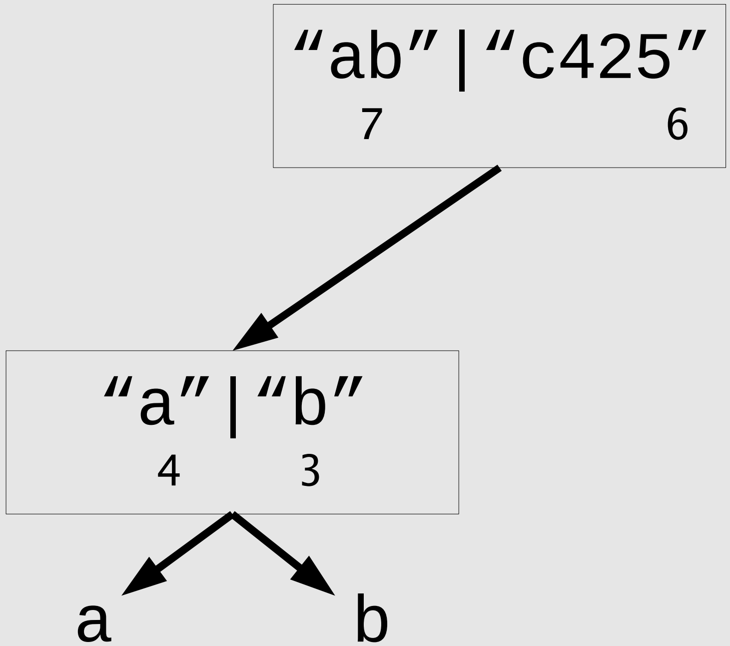
letter	count
a	4
b	3
c	2
4	2
2	1
5	1



Shannon-Fano Coder

"2ab4a5cabbc4a"

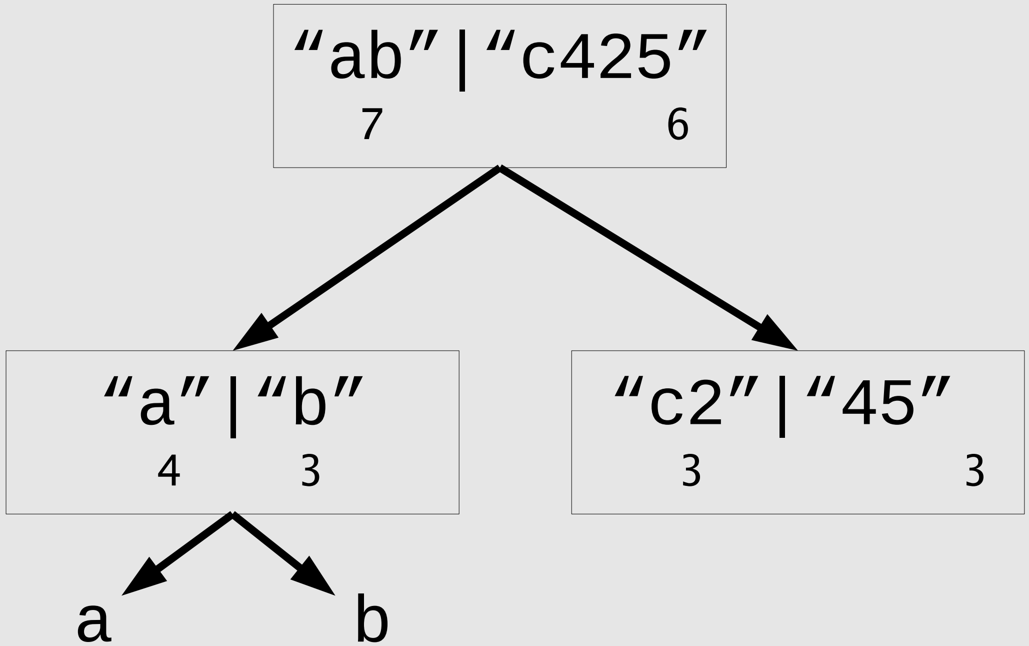
letter	count
a	4
b	3
c	2
4	2
2	1
5	1



Shannon-Fano Coder

"2ab4a5cabbc4a"

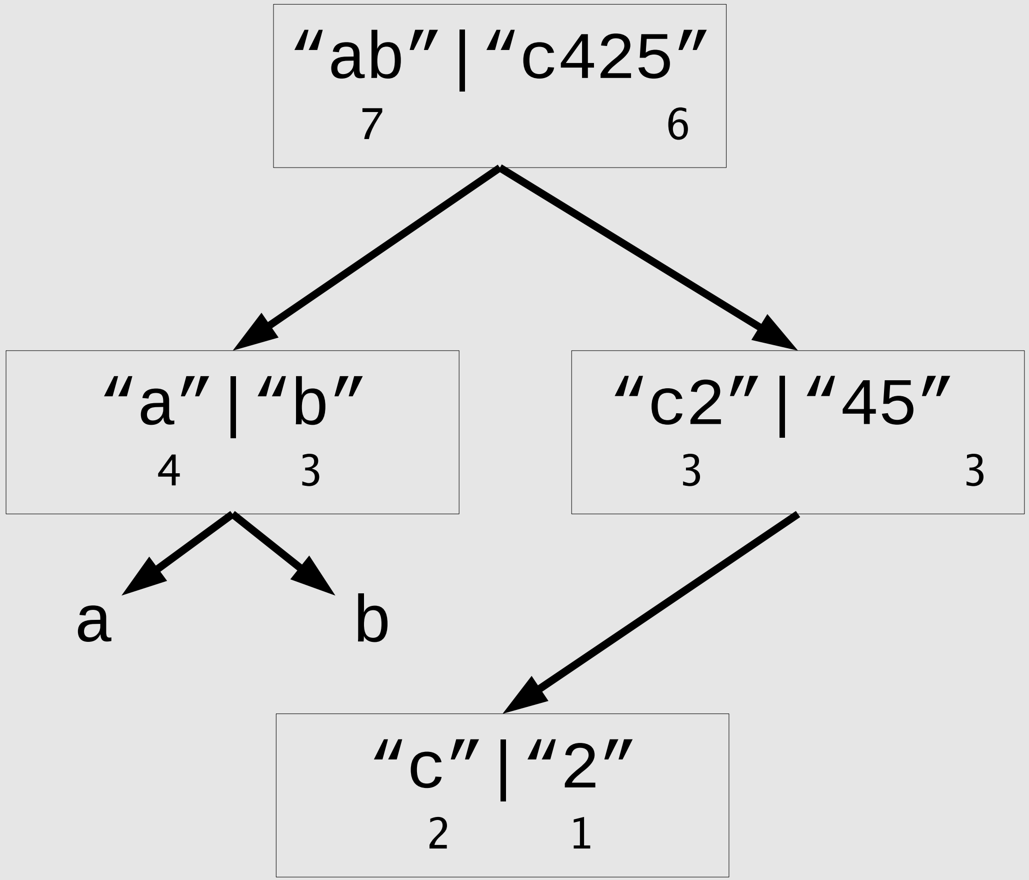
letter	count
a	4
b	3
c	2
4	2
2	1
5	1



Shannon-Fano Coder

“2ab4a5cabbc4a”

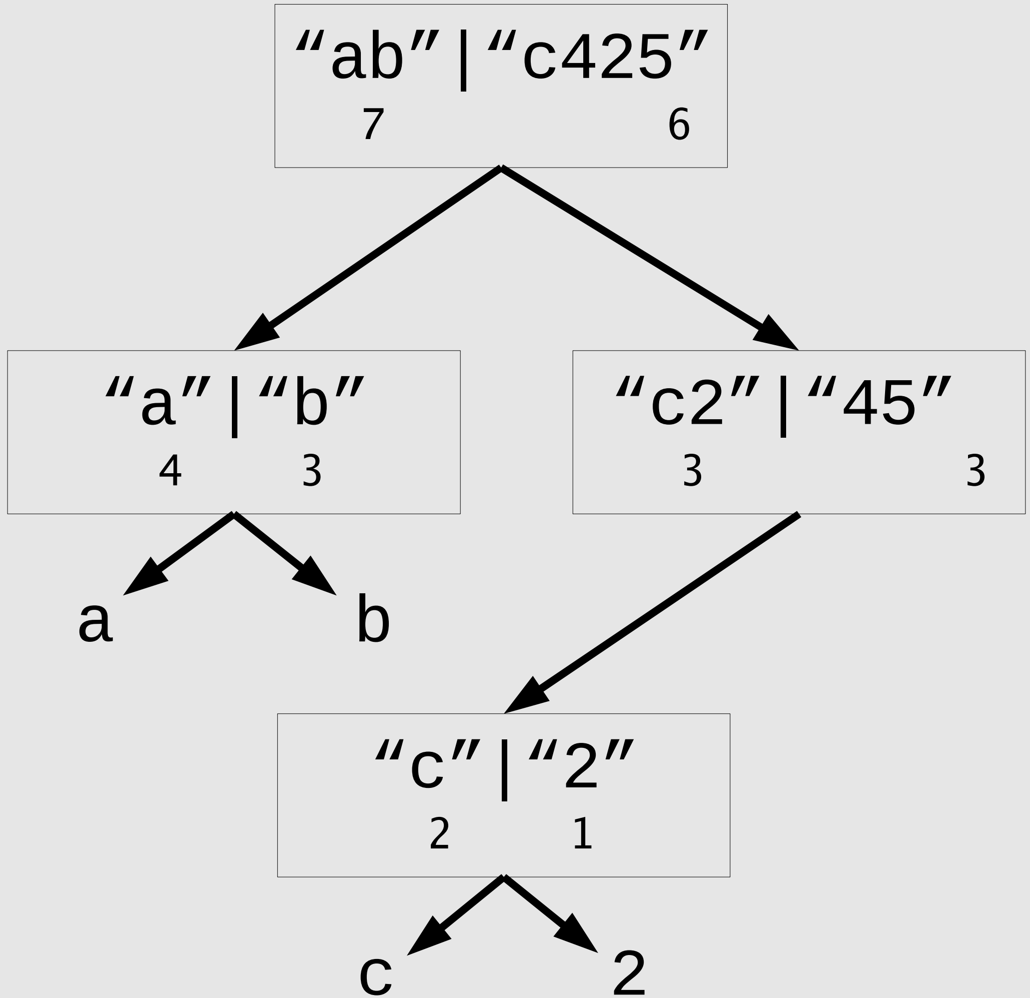
letter	count
a	4
b	3
c	2
4	2
2	1
5	1



Shannon-Fano Coder

"2ab4a5cabbc4a"

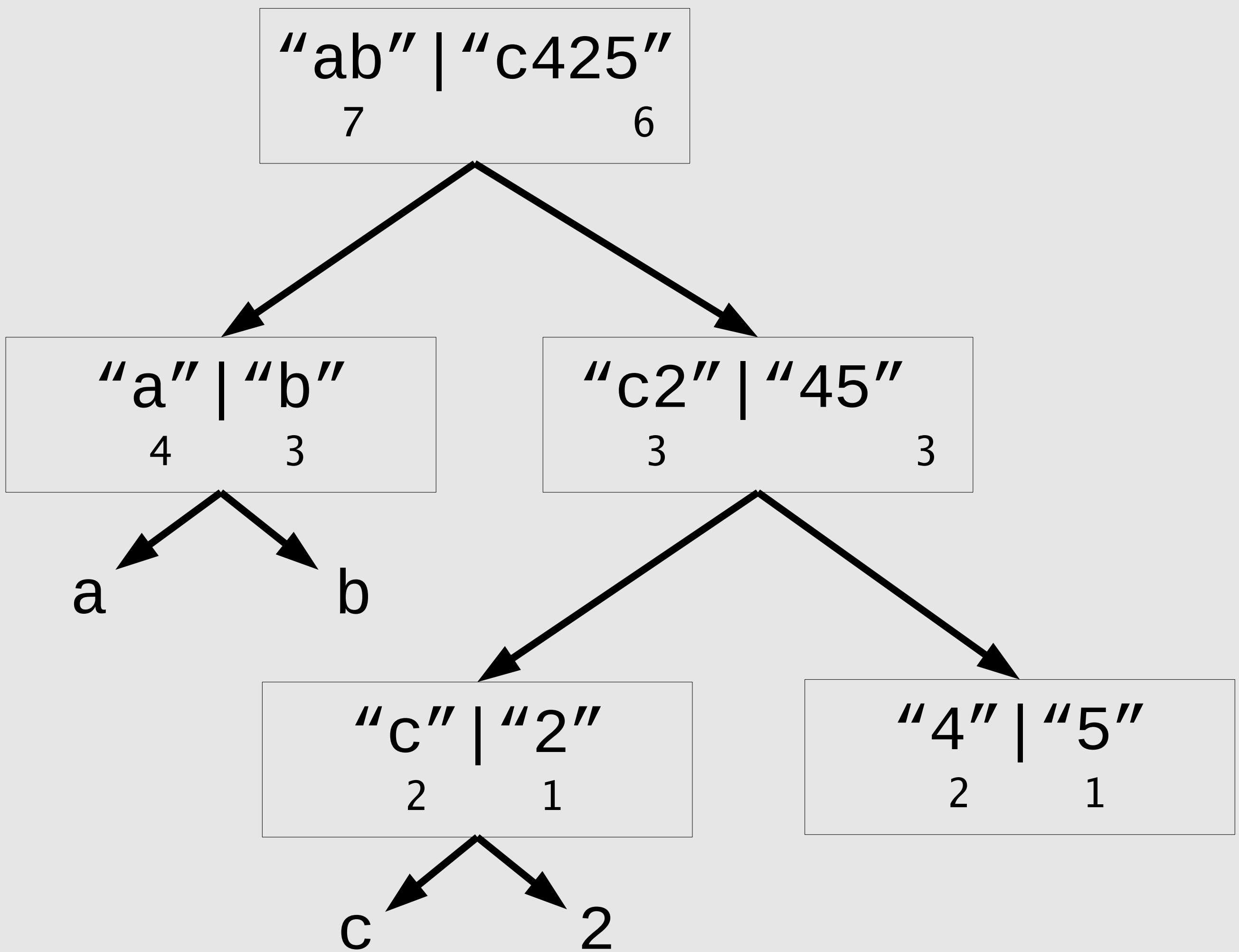
letter	count
a	4
b	3
c	2
4	2
2	1
5	1



Shannon-Fano Coder

"2ab4a5cabbc4a"

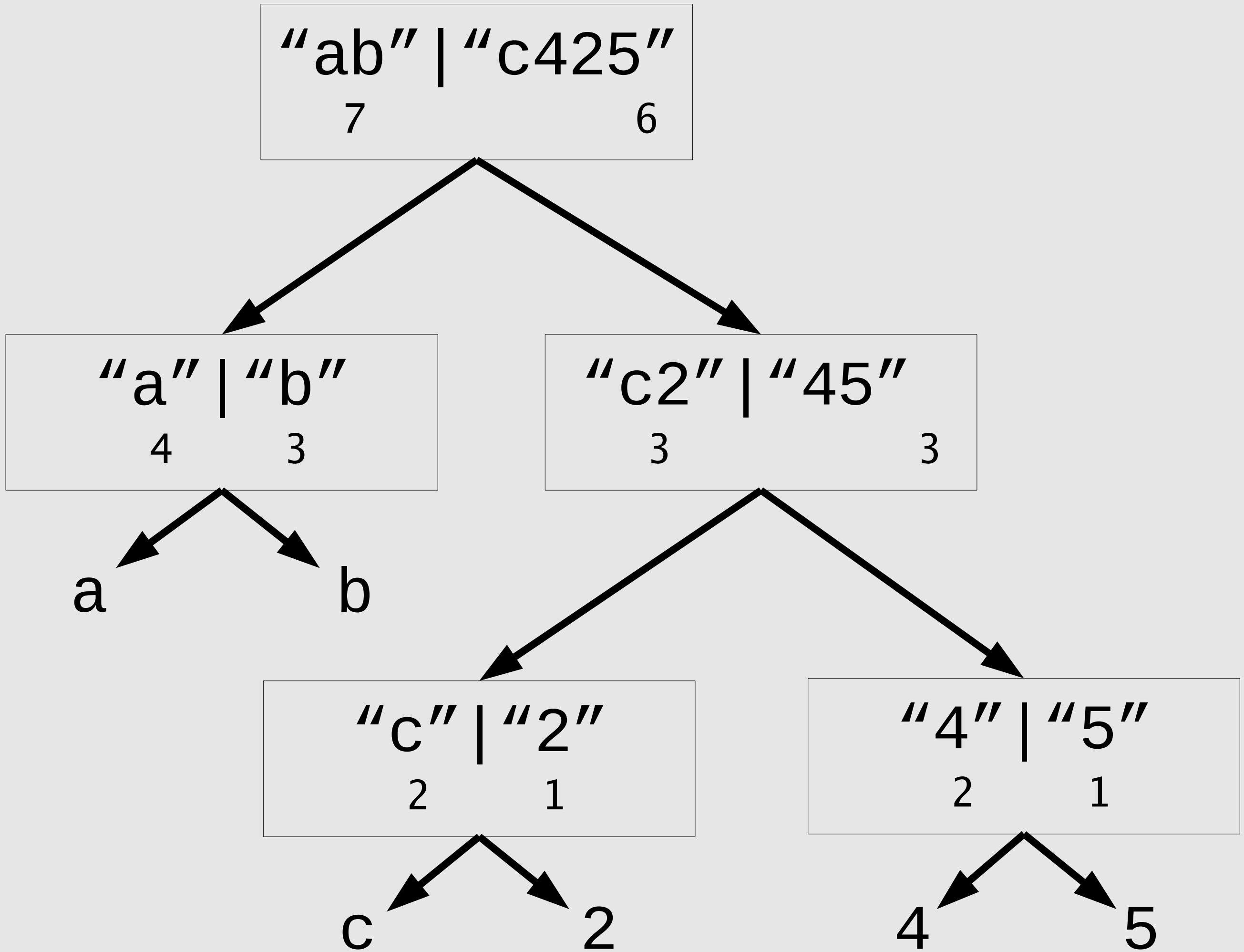
letter	count
a	4
b	3
c	2
4	2
2	1
5	1



Shannon-Fano Coder

"2ab4a5cabbc4a"

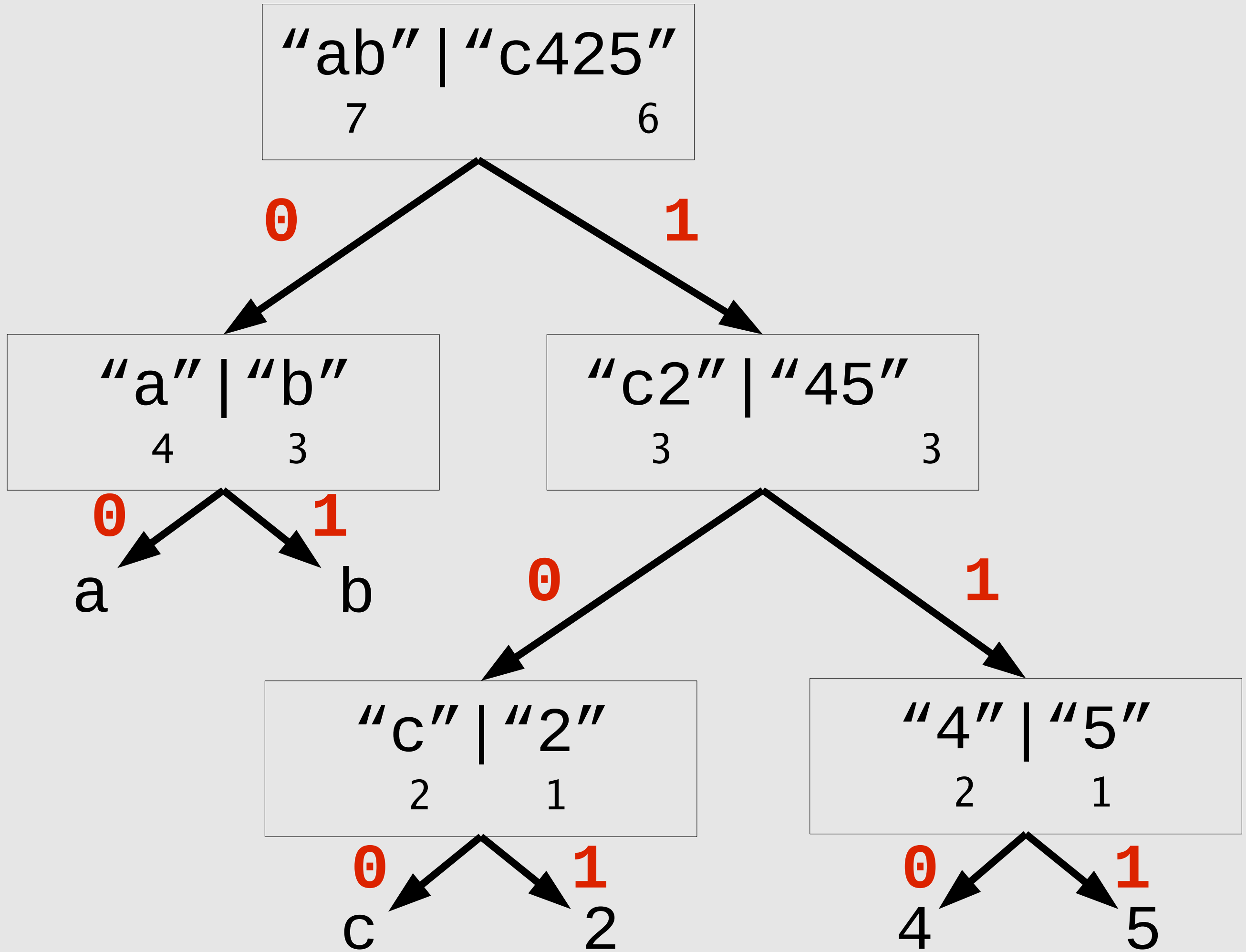
letter	count
a	4
b	3
c	2
4	2
2	1
5	1



Shannon-Fano Coder

"2ab4a5cabbc4a"

letter	count
a	4
b	3
c	2
4	2
2	1
5	1



Shannon-Fano Coder – Results

input message: “2ab4a5cabbc4a”

order	l	count	p(l)	H(l)	code(l)	bits(code(l))	∑ bits
1	a	4	0.30	1.70	**00**	2	8
2	b	3	0.23	2.11	**01**	2	6
3	c	2	0.15	2.70	**100**	3	6
4	4	2	0.15	2.70	**110**	3	6
5	2	1	0.07	3.70	**101**	3	3
6	5	1	0.07	3.70	**111**	3	3
				H=2.76			32

Shannon-Fano Coder – Results

input message: “2ab4a5cabbc4a”

l	a	b	c	4	2	5
kód(l)	00	01	100	110	101	111

2 a b 4 a 5 c a b b c 4 a

101 00 01 110 00 111 100 00 01 01 100 110 00

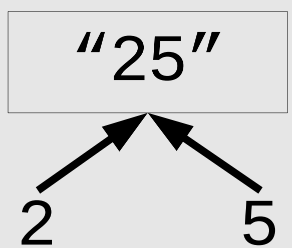
10100011100011110000010110011000

... each letter is represented by an integer bit count

Huffman Coder

“2ab4a5cabbc4a”

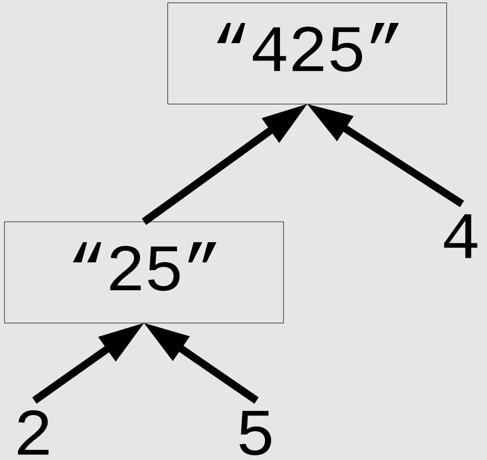
L	n	L	n
a	4	a	4
b	3	b	3
c	2	c	2
4	2	4	2
2	1	25	2
5	1		



Huffman Coder

"2ab4a5cabbc4a"

L	n	L	n
a	4	a	4
b	3	425	4
c	2	b	3
4	2	c	2
25	2		

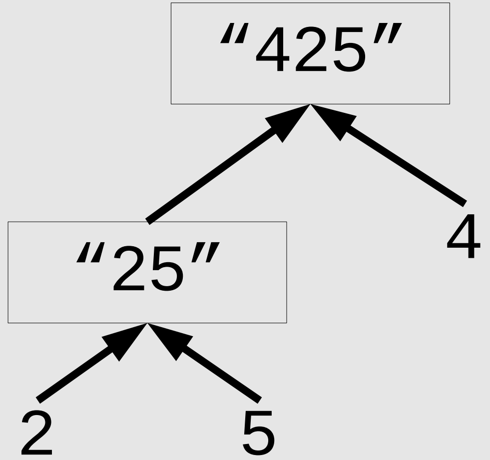


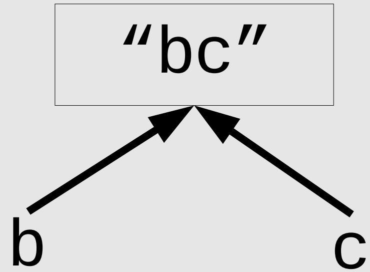
Huffman Coder

“2ab4a5cabbc4a”

L	n
a	4
425	4
b	3
c	2

L	n
bc	5
a	4
425	4



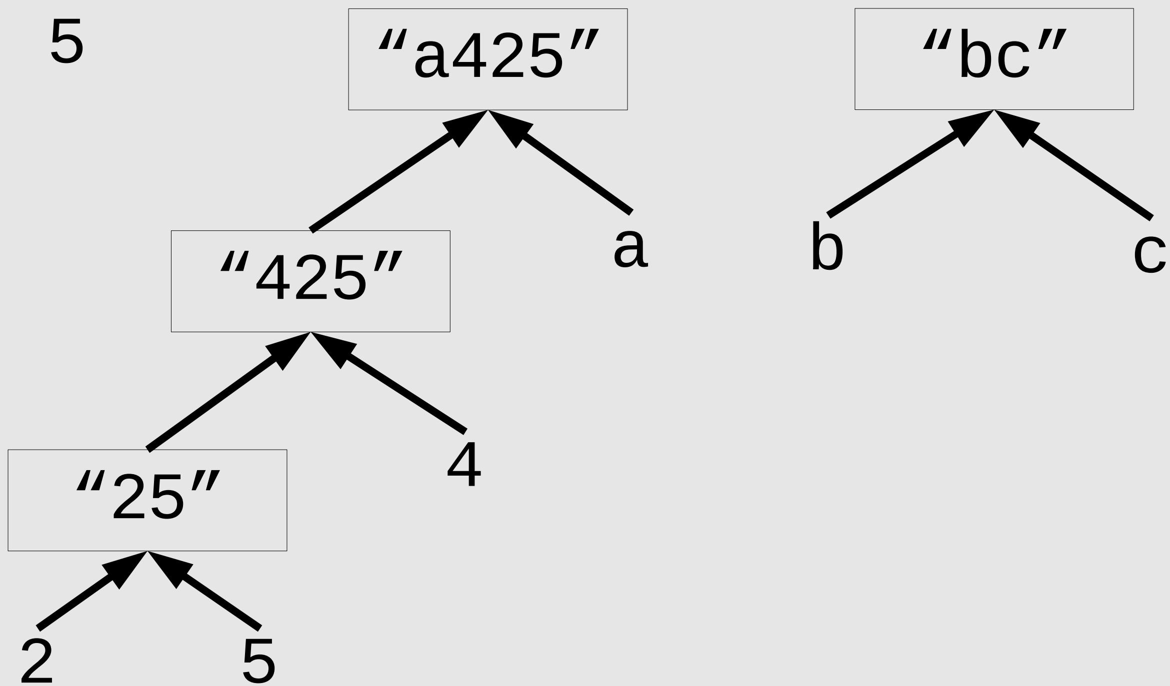


Huffman Coder

"2ab4a5cabbc4a"

L	n
bc	5
a	4
425	4

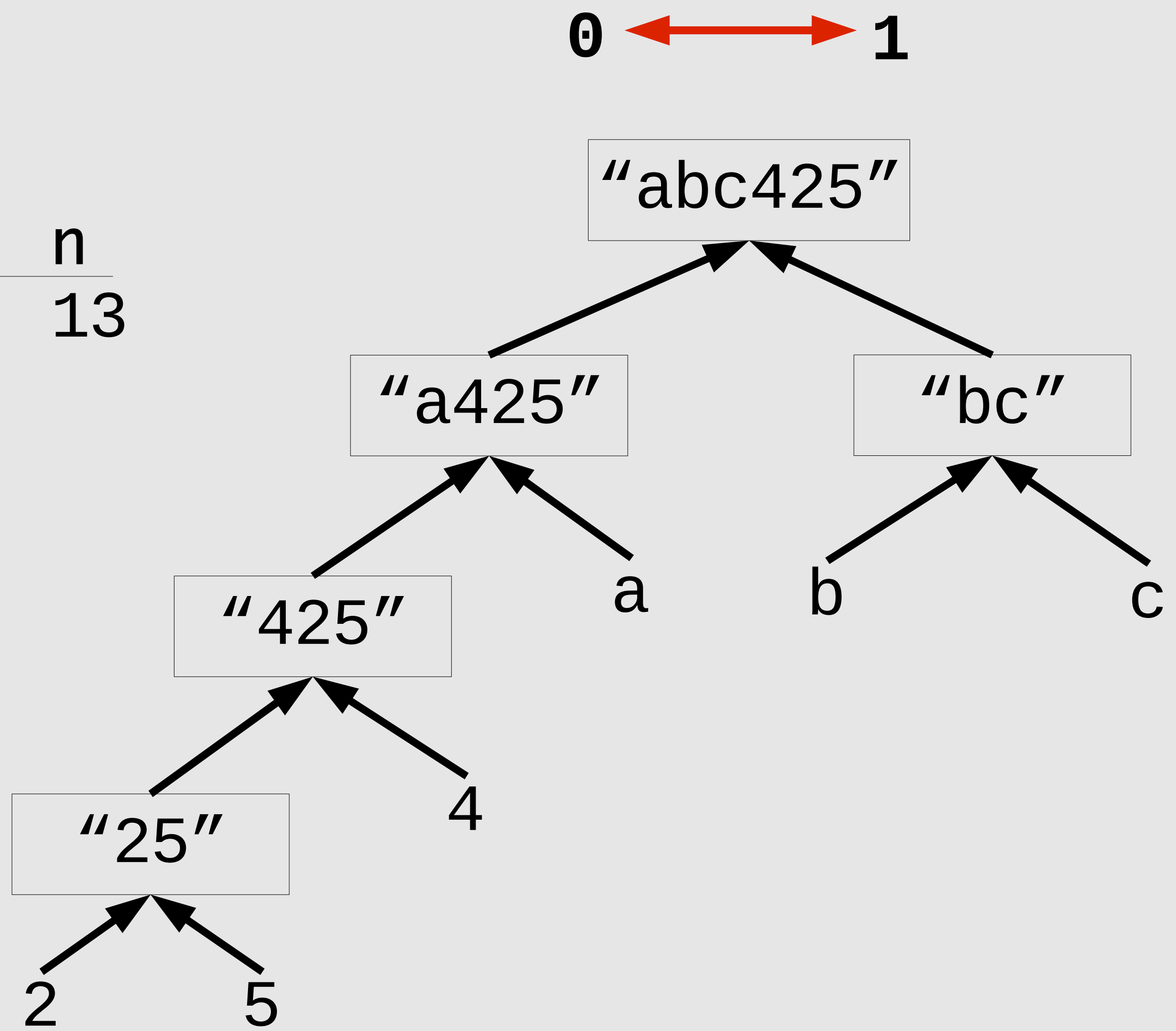
L	n
a425	8
bc	5



Huffman Coder

"2ab4a5cabbc4a"

L	n	L	n
a425	8	abc425	13
bc	5		



Huffman Coder – Results

input message: “2ab4a5cabbc4a”

pořadí	l	četnost	p(l)	H(l)	kód(l)	bitů(kód(l))	∑ bitů
1	a	4	0.30	1.70	**01**	2	8
2	b	3	0.23	2.11	**10**	2	6
3	c	2	0.15	2.70	**11**	2	4
4	4	2	0.15	2.70	**001**	3	6
5	2	1	0.07	3.70	**0000**	4	4
6	5	1	0.07	3.70	**0001**	4	4
				H=2.76			32

Huffman Coder – Results

input message: “2ab4a5cabbc4a”

l	a	b	c	4	2	5
code(l)	**01**	**10**	**11**	**001**	**0000**	**0001**

2 a b 4 a 5 c a b b c 4 a

0000 01 10 001 01 0001 11 01 10 10 11 001 01

00000110001010001110110101100101

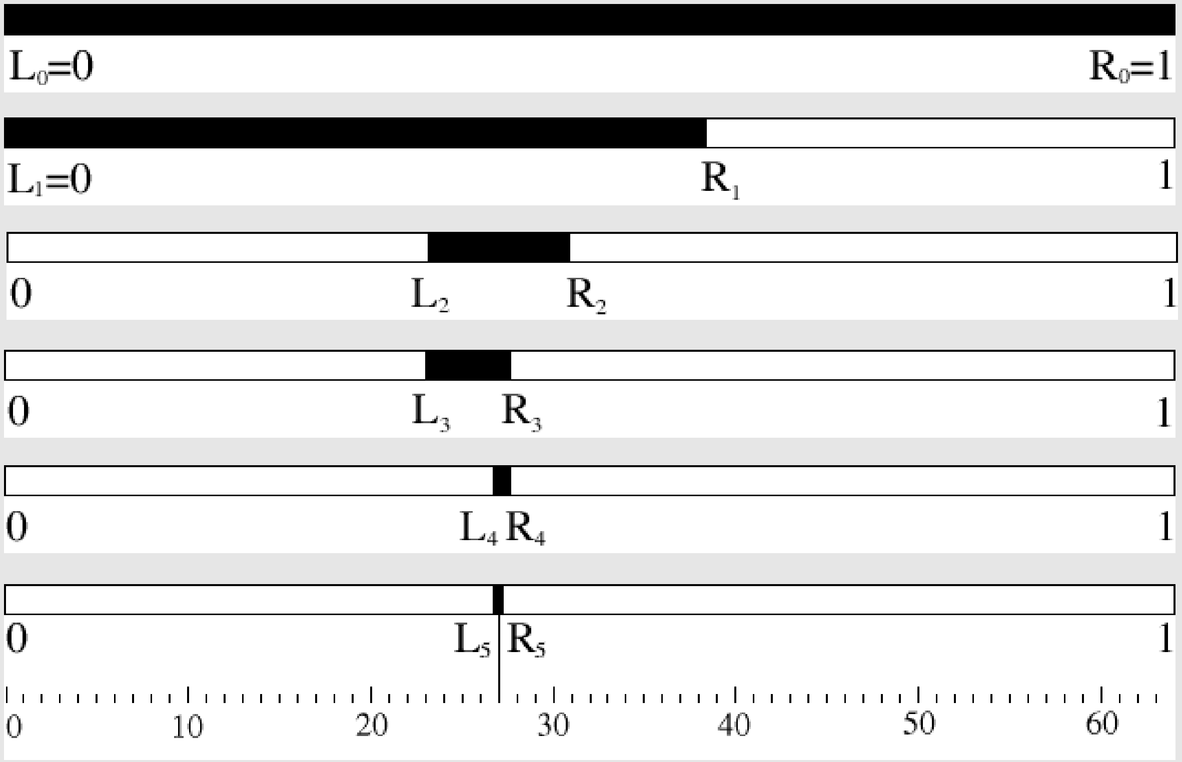
... each letter has to have an integer bit count

... quick, *O(N* log*N)*

Arithmetic Coder

input message: “abaca”

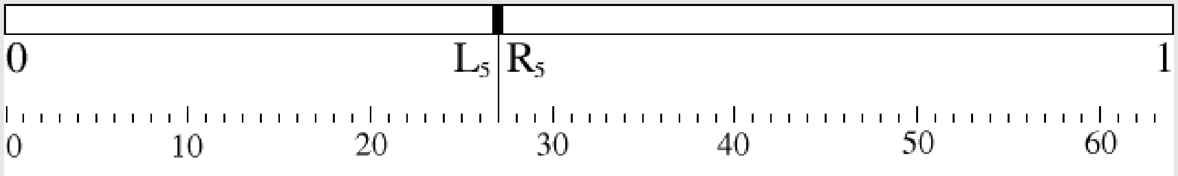
$\mathtt{a} = \langle 0, \frac{3}{5}), \mathtt{b} = \langle \frac{3}{5}, \frac{4}{5}), \mathtt{c} = \langle \frac{4}{5}, 1).$



Arithmetic Coder

input message: “abaca”

$\mathtt{a} = \langle 0, \frac{3}{5}), \mathtt{b} = \langle \frac{3}{5}, \frac{4}{5}), \mathtt{c} = \langle \frac{4}{5}, 1).$



i	z_i	L_i	R_i	$l = R_i - L_i$	$L(z_i)$	$R(z_i)$	$L_{i+1} = L_i + lL(z_i)$	$R_{i+1} = L_i + lR(z_i)$
0	a	0	1	1	0	$\frac{3}{5}$	0	$\frac{3}{5}$
1	b	0	$\frac{3}{5}$	$\frac{3}{5}$	$\frac{3}{5}$	$\frac{4}{5}$	$\frac{9}{25}$	$\frac{12}{25}$
2	a	$\frac{9}{25}$	$\frac{12}{25}$	$\frac{3}{25}$	0	$\frac{3}{5}$	$\frac{9}{25}$	$\frac{54}{125}$
3	c	$\frac{9}{25}$	$\frac{54}{125}$	$\frac{9}{125}$	$\frac{4}{5}$	1	$\frac{261}{625}$	$\frac{54}{125}$
4	a	$\frac{261}{625}$	$\frac{54}{125}$	$\frac{9}{625}$	0	$\frac{3}{5}$	$\frac{261}{625}$	$\frac{1332}{3125}$

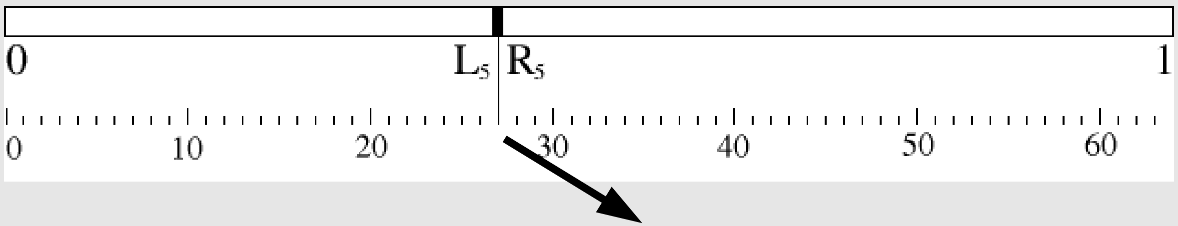
$\langle \frac{261}{625}, \frac{1332}{3125})$

$$p_a p_b p_a p_c p_a = \frac{3}{5}\frac{1}{5}\frac{3}{5}\frac{1}{5}\frac{3}{5} = \frac{1332}{3125} - \frac{261}{625} = \frac{27}{3125}.$$

Arithmetic Coder

input message: “abaca”

$\mathtt{a} = \langle 0, \frac{3}{5}), \mathtt{b} = \langle \frac{3}{5}, \frac{4}{5}), \mathtt{c} = \langle \frac{4}{5}, 1).$



... output of the arithmetic coder is 27 (**11011**)

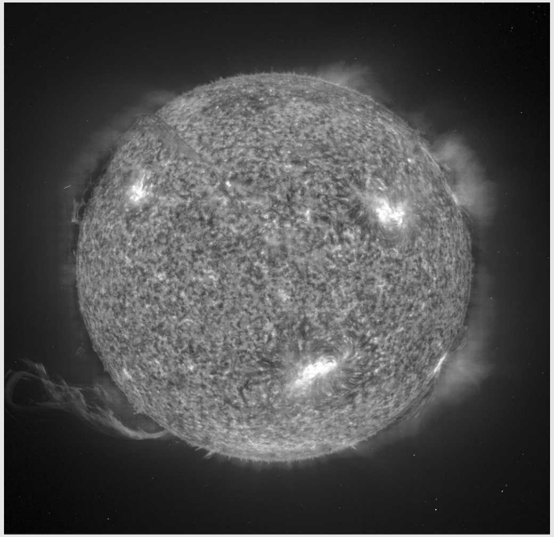
... this only number defines message “abaca” unambiguously

... it is necessary to store number of encoded letters as well as a histogram

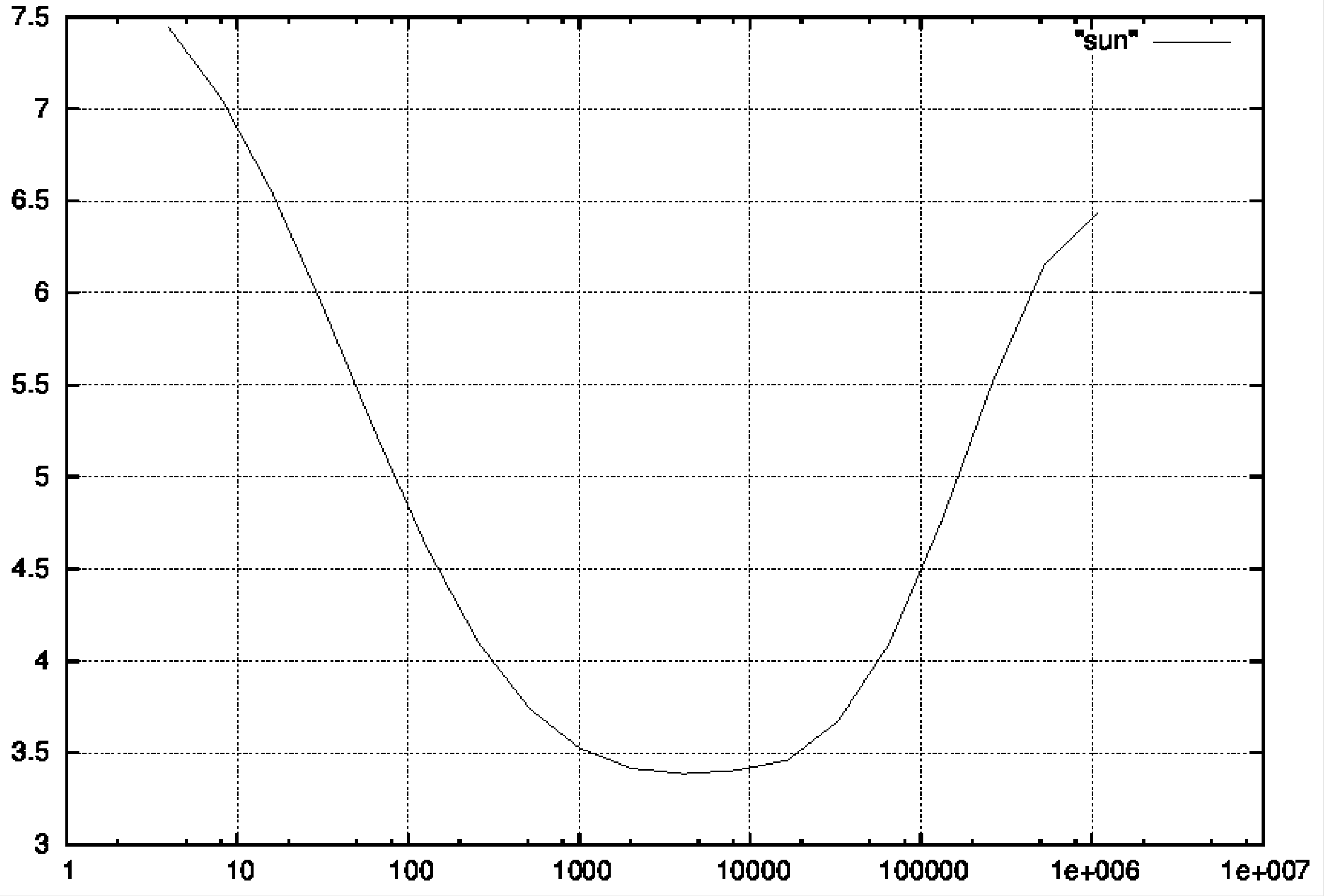
... it doesn't use integer counts for single letter

Adaptive scheme?

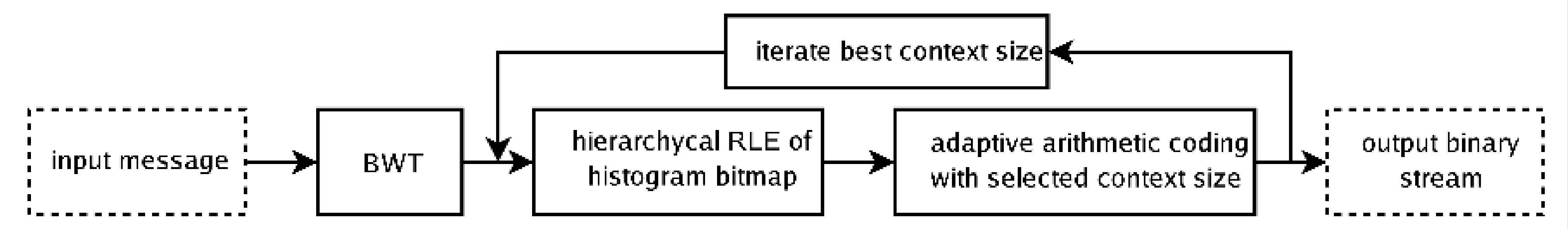
Arithmetic Coder – Finite Context



počet pixelů: 1 048 576
literálů: 135
klasicky: 6.45 b/l
velikost: 845 571 B
kontextově: 3.38 b/l
velikost: 444 014 B



Combined Methods



Name	File size [B]	Entropy [b]	Algorithm
GIF	700 206	5.342	LZW
TIFF	621 772	4.743	LZ77, Huffman
GNU zip v1.3.3	608 016	4.638	LZ77, Huffman
UNIX compress v4.2.4	604 225	4.609	LZW
RAR v3.3 beta 1	566 518	4.322	unknown
PNG	542 779	4.141	LZ77, Huffman
bzip2 v1.0.2	470 925	3.592	BWT, Huffman
proposed algorithm	444 014	3.387	BWT, Arit.

Compression Methods Discussed

- Sequential Coders
  - RLE coding
  - BWT
  - Lempel-Ziv coding
    - LZ77
    - LZW
- Statistical Coders
  - Entropy
  - Shannon-Fano Coding
  - Huffman Coding
  - Arithmetic Coding