

# Byte aligned codes

Prefix free codewords, no codeword is the prefix of another

Today: byte aligned, each codeword is a sequence of bytes  
much faster especially in decompression

1. Variable-byte uses 7 bits data 1 bit control
2. Generalization: (s,c) dense codes
3. There is a variant offering random access to single codewords, but unfortunately we don't have time for that

Example. Variable byte encoding of 1234: we split the binary representation into blocks of 7 bits; we add a continuation bit if the block is not the last in the codeword.

$$1234 = ( \text{0001001 } \text{1010010} )_2 \\ ( \text{10001001 } \text{01010010} )$$

VB encoding = (137, 82) [big endian representation]

It is immediate to see that:

values in [0,127] -> 1 byte  
[128, 128\*\*2-1] -> 2 bytes  
[128\*\*2, 128\*\*3 -1] -> 3 bytes  
and so on....

## Variants:

### 1) 3 bits digits, 1 bit continuation

each codeword is made of blocks of 4 bits (aka nibbles)

Number of bits x integer

values in  $[0,7]$  -> 4 bits

$[8,64-1]$  -> 8 bits

$[2^{**6}, 2^{**9}-1]$  -> 12 bits

and so on ...

### 2) 15 bits digits, 1 bit continuation.

each codeword is made of blocks of 16 bits

values in  $[0, 2^{**15} - 1]$  -> 2 bytes

$[2^{**15}, 2^{**30}-1]$  -> 4 bytes

And so on with blocks of  $23+1$ ,  $31+1$ ,  $39+1$ , ... bits

The larger the block the more efficient the encoding of large integers (and less efficient the encoding of small integers)

NEXT TOPIC: *(s,c)-dense codes (byte aligned)*

(s,c) dense codes: generalization of VB encoding

in VB encoding

values [128,255] are Continuation values

values [0,127] are Stop values

we have 128 C-values and 128 S-values

Why not using

100 C-values and 156 S-values?

Or any other combination with  $C+S = 256$ ?

This is what we do in (s,c) codes: the encoding is a sequence of bytes: values [0,s-1] are stop values, values in [s,c+s-1] are continuation values

*Simplified example: 3-bit codes (s+c=8)*

| Values | $s = c = 4$ | $s = 6, c = 2$                      |
|--------|-------------|-------------------------------------|
| 0      | 000         | 000                                 |
| 1      | 001         | 001                                 |
| 2      | 010         | 010                                 |
| 3      | 011         | 011                                 |
| 4      | 100 000     | 100                                 |
| 5      | 100 001     | 101                                 |
| 6      | 100 010     | 110 000                             |
| 7      | 100 011     | 110 001                             |
| 8      | 101 000     | 110 010                             |
| 9      | 101 001     | 110 011                             |
| 10     | 101 010     | 110 100                             |
| 11     | 101 011     | 110 101                             |
| 12     | 110 000     | 111 000                             |
| 13     | 110 001     | 111 001                             |
| 14     | 110 010     | 111 010                             |
| 15     | 110 011     | 111 011                             |
| 16     | 111 000     | 111 100                             |
| 17     | 111 001     | 111 101                             |
| 18     | 111 010     | $\overset{C}{111} \overset{S}{101}$ |
| 19     | 111 011     |                                     |

|            | $s = c = 4$ | $s = 6, c = 2$ |
|------------|-------------|----------------|
| $s$ - from | 000         | 000            |
| $s$ - to   | 011         | 101            |
| $c$ - from | 100         | 110            |
| $c$ - to   | 111         | 111            |
| $s$        | 4 (4)       | 6 (6)          |
| $cs$       | 16 (20)     | 12 (18)        |
| $c^2s$     | 64 (86)     | 24 (42)        |

FIGURE 11.6: Example of  $(s, c)$ -encoding using two different values for  $s$  and  $c$ . In brackets we show the starting value for all codewords having a given composition.

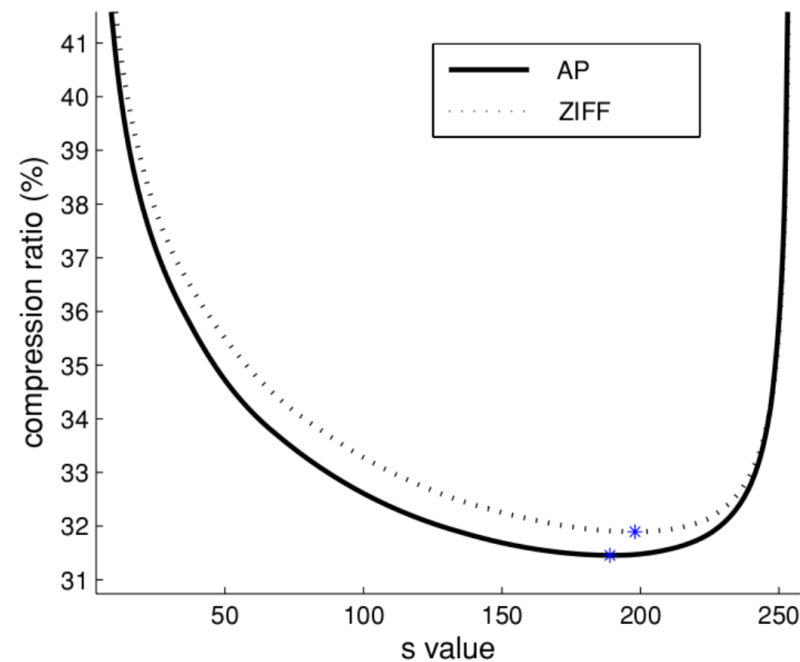
See the coding and decoding procedures in the python notebook:

```
sc_encode(n,s,c)      # encodes a single integer  
sc_decode(x,s,c)      # decodes a single integer  
sc_encode_array(a,s,c) # encodes an array of integers  
sc_decode_array(y,s,c) # decodes an array of integers
```

The next step is to select the  $(s,c)$  pair providing the more effective dense code for a given input array

Assuming we want a byte aligned code,  $c=256-s$  so we have a single free parameter.

Experiments show that the encoding size as a function of  $s$  has a unique minimum:



A function  $f(x)$  with a unique minimum, which is always decreasing before the minimum, and always increasing after the minimum is called **unimodal**.

Assume we know the minimum  $x^*$  is in  $(a,b)$ , consider any pair of points  $x_1 < x_2$  in  $(a,b)$  then:

if  $f(x_1) > f(x_2)$  then  $x^* \in (x_1, b)$

if  $f(x_1) < f(x_2)$  then  $x^* \in (a, x_2)$

if  $f(x_1) = f(x_2)$  then  $x^* \in (x_1, x_2)$

We can select  $x_1, x_2$  so that the interval  $[a,b]$  is split into 3 equal parts, so that at each iteration the size of the interval shrinks by at least a factor  $2/3$ , eg:

$$[a \text{ — } x_1 \text{ — } x_2 \text{ — } b]$$

See the function `find_minimum_encoding()` for an example

## *Not only integer sequences: word based compression, another application of integer coding*

We want to parse an input text into words (discarding punctuation) and assign to each word a distinct prefix-free codeword

The set of symbols is the vocabulary that can get very large: as new documents are added the vocabulary continues to grow

We could use Huffman algorithm to optimally assign codewords to words

In practice it works well to sort the words by frequency and assign to the  $i$ -th most frequent word the codeword for the integer  $i$  in a proper prefix-free integer compression scheme.

Note that more frequent words are associated to small ranks and therefore to shorter codewords: not as good as Huffman but much simpler and faster (especially if we use a byte aligned code)