A Crash Course on Data Compression

2. Integer Codes

Giulio Ermanno Pibiri

ISTI-CNR, giulio.ermanno.pibiri@isti.cnr.it

@giulio_pibiri

@jermp

Overview

- Binary and Unary
- Gamma and Delta
- Golomb-Rice
- Exponential Golomb
- Fibonacci
- Variable-Byte
- Effectiveness, Information content, Entropy, Kraft-McMillan inequality

The *Static* Integer Coding Problem

- **Problem.** We are given an integer $x > 0$, and we have to design an algorithm — a *code* — that represents x in *as few as possible bits*.

- **Codeword.** The bit-string representing x according to the chosen code is called the *codeword* of x, and indicated with $C(x)$.

- A message $L = [x_1, \ldots, x_n]$ consisting of n integers will be coded as the concatenation of the codewords assigned to $x_1, \ldots, x_n$, i.e., $C(x_1) \cdots C(x_n)$.

- **Static codes.** The codes we study in this module are called *static* because they always assign the same codeword $C(x)$ to the integer x, *regardless* the message L to be coded.

Binary

- **Binary string of fixed length.** We indicate with $bin(x, k)$ the representation of $0 \leq x < 2^k$ using k bits. If we just write $bin(x)$, we assume k is equal to $\lceil \log_2(x+1) \rceil$ which is the *minimum number of bits necessary to represent* x.

- **Binary codewords.** Since we assume $x > 0$, we say that $B(x) = bin(x-1)$ is the codeword assigned to x by the binary code.

- **Lower bound.** The size of any codeword $C(x)$, for $x > 0$, is: $|C(x)| \geq \lceil \log_2(x) \rceil = |bin(x-1)| = |B(x)|$.

x	B(x)
1	0
2	1
3	10
4	11
5	100
6	101
7	110
8	111

A First Attempt

- **Idea.** Since $|C(x)| > |B(x)|$ for any code C, given a message $L = [x_1, \ldots, x_n]$, let's encode L as $B(x_1)\cdots B(x_n)$.

  Example. $L = [3,5,2,6,12,8] \rightarrow$ `10.100.1.101.1011.111`

x	B(x)
1	`0`
2	`1`
3	`10`
4	`11`
5	`100`
6	`101`
7	`110`
8	`111`

A First Attempt

- **Idea.** Since $|C(x)| > |B(x)|$ for any code C, given a message $L = [x_1, \ldots, x_n]$, let's encode L as $B(x_1)\cdots B(x_n)$.

  Example. $L = [3,5,2,6,12,8] \rightarrow$ `10.100.1.101.1011.111`

- Ok, now that we have the message coded as `101001101011111`, we want to decode it — get $L = [3,5,2,6,12,8]$ back.

- **Q.** How?

x	B(x)
1	0
2	1
3	10
4	11
5	100
6	101
7	110
8	111

A First Attempt (Failed)

- **Idea.** Since $|C(x)| > |B(x)|$ for any code C, given a message $L = [x_1, \ldots, x_n]$, let's encode L as $B(x_1)\cdots B(x_n)$.

  Example. $L = [3,5,2,6,12,8] \rightarrow$ `10.100.1.101.1011.111`

- Ok, now that we have the message coded as `1010011011011111`, we want to decode it — get $L = [3,5,2,6,12,8]$ back.

- **Q.** How?

  A. Many possibly ways of decoding the message!
  Our code is *ambiguous*.

x	$\mathrm{B}(x)$
1	0
2	1
3	10
4	11
5	100
6	101
7	110
8	111

Unique Decodability

- **Fact.** If no codeword is *prefix* of another one, we can decode without ambiguity.
- **Prefix-free code.** A code C is said to be prefix-free when: there are no $C(x)$ and $C(y)$, with $C(y) \geq C(x)$, for which $C(x) = C(y)[1 : |C(x)|]$.
- We are only interested in prefix-free codes.

x	$B(x)$
1	0
2	1
3	10
4	11
5	100
6	101
7	110
8	111

The binary code is not prefix-free.

x	$C(x)$
1	00
2	01
3	100
4	101
5	1100
6	1101
7	11100
8	11101

An example prefix-free code.

Unary

- **Idea.** Use the bit `1` for data; the bit `0` as a reserved symbol to delimit the codewords.

- Represent $x > 0$ as $U(x) = 1^{x-1}0$, i.e., a run of $(x-1)$ 1s plus a final `0`. Therefore, $|U(x)| = x$.

x	U(x)
1	0
2	10
3	110
4	1110
5	11110
6	111110
7	1111110
8	11111110

Unary

- **Idea.** Use the bit `1` for data; the bit `0` as a reserved symbol to delimit the codewords.

- Represent $x > 0$ as $U(x) = 1^{x-1}0$, i.e., a run of $(x - 1)$ 1s plus a final `0`. Therefore, $|U(x)| = x$.

- The code is only good for (very) small integers.

Example 1.
$L = [3,5,2,6,12,8] \rightarrow$ `110.11110.10.111110.111111111110.11111110`

Example 2. $U(234) =$
```
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111110
```
234 bits for a single integer!

x	U(x)
1	0
2	10
3	110
4	1110
5	11110
6	111110
7	1111110
8	11111110

Gamma and Delta

Elias, 1975

- **Idea.** Before writing $bin(x)$, specify how many bits we have in $bin(x)$.
- **Gamma.** Write $b = |bin(x)|$ using Unary, followed by the $b - 1$ least significant bits of $bin(x)$.
  We have $|\gamma(x)| = 2|bin(x)| - 1$ bits, roughly a factor of 2 away from the optimum.
- **Q.** Why the $b - 1$ least significant bits of x and not b?
  A. Because the integers that have a minimum binary length of b bits are those in the range $[2^{b-1}, 2^b - 1]$ for which the most significant bit is always 1, so it is redundant.
- **Delta.** Replace the Unary part of Gamma, $U(b)$, with $\gamma(b)$ because $U(b)$ can be very large for big integers.
  We have $|\delta(x)| = |\gamma(|bin(x)|)| + |bin(x)| - 1$ bits, roughly a factor of $(1 + o(1))$ away from the optimum.

x	$\gamma(x)$	$\delta(x)$
1	`0.`	`0.`
2	`10.0`	`100.0`
3	`10.1`	`100.1`
4	`110.00`	`101.00`
5	`110.01`	`101.01`
6	`110.10`	`101.10`
7	`110.11`	`101.11`
8	`1110.000`	`11000.000`

Minimal Binary

- Suppose we have to assign binary codewords to all the integers $x \in [0,b)$ where $b \leq 2^c$, for some $c \geq 0$. (We can assume $c = \lceil \log_2 b \rceil$.)
- Then $2^c - b$ codewords can be made 1 bit *shorter* without losing unique decodability, using the following “remapping” trick.
- If $x < 2^c - b$, then assign codeword $bin(x, c-1)$. Otherwise, assign codeword $bin(x + 2^c - b, c)$.
- Decoding is simple. Always read $c - 1$ bits as the quantity x: if $x < 2^c - b$, then return x; otherwise fetch another bit y and return $x' = ((x \ll 1) \,|\, y) - (2^c - b)$.

$b = 5$ and $c = 3$

x	$bin(x,3)$
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

x	$bin(x,2)$
0	00
1	01
2	10

x	$bin(x+3,3)$
3	110
4	111

Golomb-Rice

Golomb, 1966 — Rice, 1971

- **Idea.** Reduce the magnitude of x by division.
- The Golomb code makes use of an integer parameter $b > 1$.
- $G_b(x)$ consists in coding the quotient $q = \lfloor (x-1)/b \rfloor$ and the reminder $r = x - q \cdot b - 1$.
- The quantity $q + 1$ is coded in Unary; r is coded as $bin(r, \lceil \log_2 b \rceil)$. (Or in Minimal Binary in the interval $[0,b)$.)
- The Rice code is a Golomb code for which $b = 2^k$ for some $k > 0$. (Better decoding speed when b is a power of 2.)

x	$G_2(x)$
1	0.0
2	0.1
3	10.0
4	10.1
5	110.0
6	110.1
7	1110.0
8	1110.1

Exponential Golomb

Teuhola, 1978

- **Idea.** Use many Golomb codes with different parameters b.
- Define a vector of “buckets”:

$$B = \left[0, 2^k, \sum_{i=0}^{1} 2^{k+i}, \sum_{i=0}^{2} 2^{k+i}, \sum_{i=0}^{3} 2^{k+i}, \ldots\right], \text{ for some } k \geq 0.$$

- Encode an integer x as the index of bucket where it belongs to, plus an offset relative to the bucket.
- The index is an integer $h \geq 1$ such that $B[h] < x \leq B[h+1]$ and is coded in Unary, whereas the offset is the quantity $x - B[h] - 1$ and coded as $bin(x - B[h] - 1, \log_2(B[h+1] - B[h]))$.

x	$G_2(x)$	$ExpG_2(x)$
1	0.0	0.00
2	0.1	0.01
3	10.0	0.10
4	10.1	0.11
5	110.0	10.000
6	110.1	10.001
7	1110.0	10.010
8	1110.1	10.011

Fibonacci

Fraenkel and Klein, 1985 — Apostolico and Fraenkel, 1987

- **Idea.** Use the Zeckendorf's theorem.

- **Zeckendorf's theorem.** *Every positive integer can be represented as the sum of some, non consecutive, Fibonacci numbers.*

- Let $F_i = F_{i-1} + F_{i-2}$ be the i-th Fibonacci number for $i > 2$, with $F_1 = 1$ and $F_2 = 2$.
  We logically define a vector $F = [F_1, F_2, F_3, \ldots] = [1{,}2{,}3{,}5{,}8{,}13{,}\ldots]$.

- If $x = F[i_1] + F[i_2] + \cdots + F[i_n]$, with $i_1 < i_2 < \cdots < i_n$, then the codeword for x is $(i_n + 1)$-bit long and is:

```
0..010..010..011
```

$$\qquad\quad i_1 \qquad\quad i_2 \qquad\quad i_n$$

x	F(x)					
1	1	**1**				
2	0	1	**1**			
3	0	0	1	**1**		
4	1	0	1	**1**		
5	0	0	0	1	**1**	
6	1	0	0	1	**1**	
7	0	1	0	1	**1**	
8	0	0	0	0	1	**1**
F_i	1	2	3	5	8	13

Variable-Byte

Thiel and Heaps, 1972

- **Idea.** Codewords are *byte-aligned* rather than bit-aligned.
- Byte-aligned codewords are useful in practice because the computer memory is allocated in chunks of bytes, not bits. Thus, working with byte-aligned codewords favours implementation simplicity and encoding/decoding speed (e.g., Single-Instruction-Multiple-Data, SIMD) — instead of compression effectiveness.
- In Variable-Byte, the binary representation of x is split in a suitable number of bytes: for each byte, 7 bits are allocated for the representation of x itself (*data* bits), and 1 bit (the *control* bit) is used to signal continuation/end of the stream of bytes.
- Variable-Byte is only effective for large integers.
- A simple variant using 4-bit payloads (3 data bits, 1 control bit) is called *nibble* coding.

Example for $x = 67822$, $bin(67822, 17)$ = `10000100011101110`.

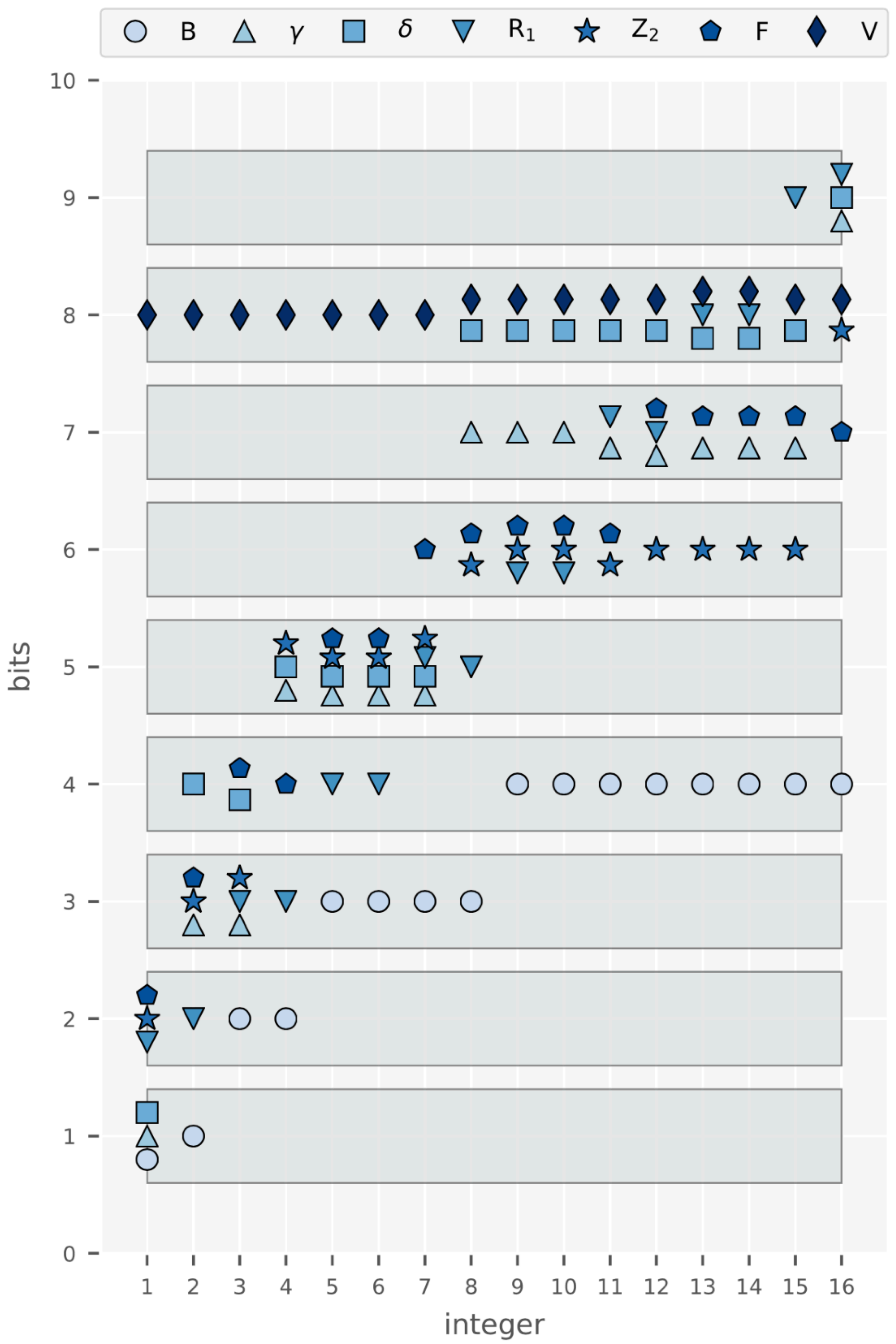
```
(1)      100. 0010001. 1101110
(2) xxxxx100.x0010001.x1101110
(3) 00000100.10010001.11101110
```

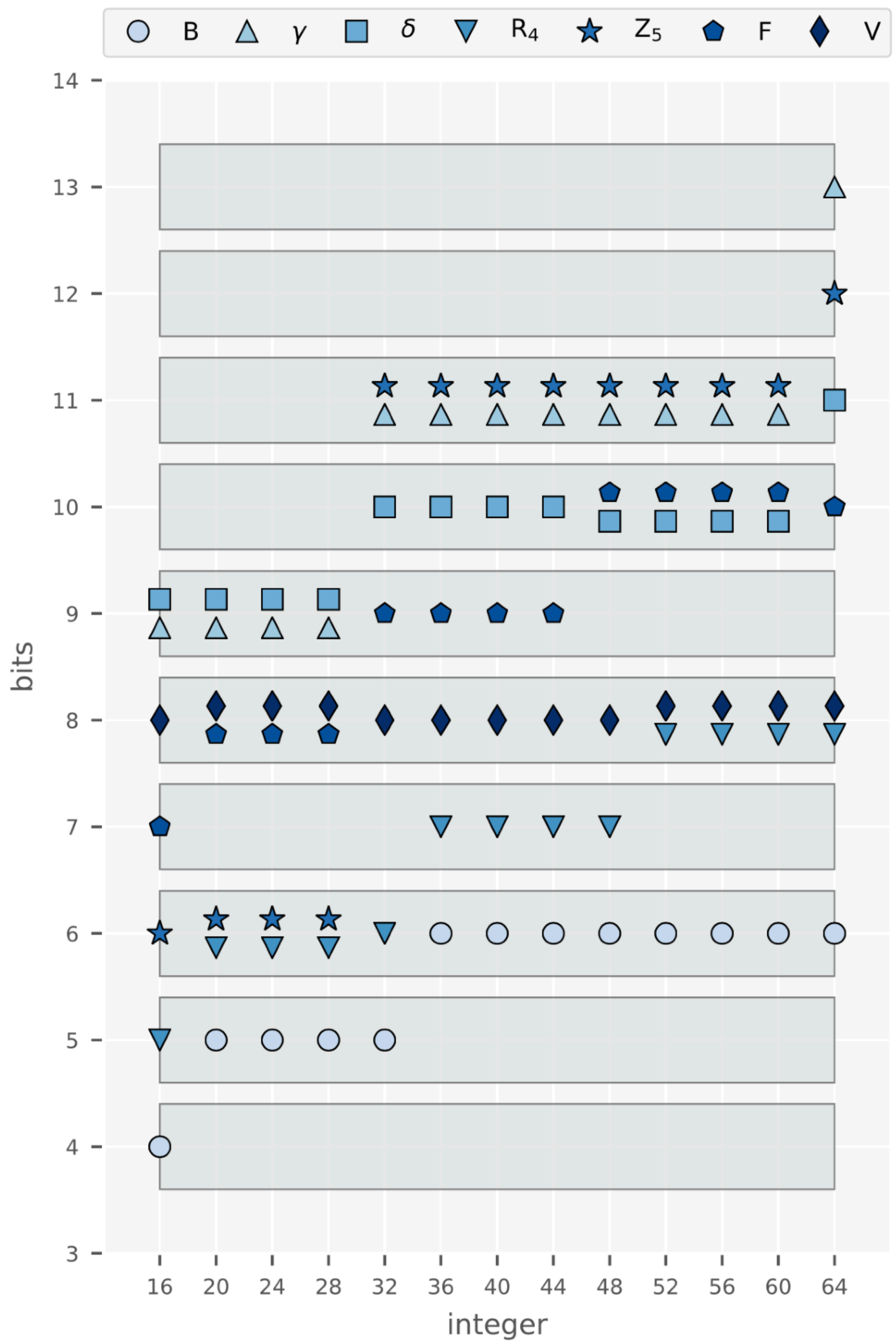
Effectiveness

Legend.

- B: binary code (lower bound)
- γ, δ: Gamma and Delta codes
- R_1: Rice code with $k = 1$
- Z_2: Zeta code (a variation of the Exponential Golomb code) with parameter $k = 2$
- F: Fibonacci-based code
- V: Variable-Byte

Q. Which code should I use?





Information Content

- **Intuition.** The effectiveness of a code depends on *how the integers are distributed* in the message L to be coded.
- Therefore, we are interested in knowing — or, at least, *estimating* — $P(x)$, the probability of occurrence of x in L.

Example 1: If the probability of small integers is very high, then the unary code is good. For example, $P(1) = 0.9$, $P(2) = 0.08$, and
$P(x > 2) = 1 - 0.9 - 0.09 = 0.01$.

Example 2: if $P(x) \approx 1/2^k \ \forall x$, for some k, then $bin(x, k)$ is optimal.

Information Content

- **Intuition.** If $P(x)$ is high, then x is very frequent in L, and it should receive a short codeword $C(x)$ — this is the so-called "golden rule" of data compression.
- So it appears that the *information content* of x is related to the its probability $P(x)$.
- **Information content.** The *information content* $I(x)$, or *self-information*, of x is defined as $\log_2(1/P(x))$ and is measured in bits.
  The *higher* $P(x)$, the *lower* the information content of x and vice versa.

Entropy

Shannon, 1949

- Given that the symbol x has information content $I(x)$, an *optimal code* T should assign a codeword $C(x)$ such that $|C(x)| = I(x) = \log_2(1/P(x))$ bits.

- **Entropy.** Therefore, we can say that:

$$H(P) = \sum_x P(x)I(x) = \sum_x P(x)\log_2\left(\frac{1}{P(x)}\right) \text{ bits}$$

is the *expected codeword length* for an optimal code T according to the distribution P.

Shannon called this quantity the *entropy of the distribution* P and it gives us a *lower bound* on the number of bits required by $C(x)$ for any code T.

Entropy — Example

- **Entropy.** $H(P) = \sum_x P(x)\log_2(1/P(x))$ bits.

- Given a message $L[1..n]$, then $P(x)$ can be estimated as $w(x)/n$ where $w(x)$ is the number of occurrences (the *weight*) of x in L.
  ($P(x) \approx w(x)/n$ is sometimes called the *self-probability* of x).

- Example for $L[1..16] = [1,3,1,1,1,5,2,1,7,3,1,2,1,1,1,1]$.
  We have $P(1) \approx 10/16$, $P(2) = P(3) \approx 2/16$, and $P(5) = P(7) \approx 1/16$.
  Then $H(P) = 2 \cdot 1/16 \cdot \log_2(16) + 2 \cdot 2/16 \cdot \log_2(16/2) + 10/16 \cdot \log_2(16/10) \approx 1.674$ bits.
  The whole message L requires, at least, $16 \cdot 1.674 = 26.784$ bits.

- For the example code on the right, the cost of the coded message is:
  $10 \cdot |C(1)| + 2 \cdot |C(2)| + 2 \cdot |C(3)| + |C(5)| + |C(7)| =$
  $10 \cdot 1 + 2 \cdot 2 + 2 \cdot 3 + 4 + 4 = 28$ bits, and the average codeword length is $28/16 = 1.75$ bits.

x	$C(x)$
1	`0`
2	`10`
3	`110`
5	`1110`
7	`1111`

En example code

Distributions

- Since it must be $|C(x)| = I(x) = \log_2(1/P(x))$ for a code to be optimal, we can invert the relation to find the distribution $P(x)$ for which the code is optimal, as $P(x) = 2^{-|C(x)|}$.

Some examples.

Unary: $P(x) = 1/2^x$

Binary: $P(x) = 1/U$, if each x is less than U and coded in $\lceil \log_2 U \rceil$ bits

Gamma: $P(x) \approx 1/(2x^2)$

Delta: $P(x) \approx 1/(2x(\log_2 x)^2)$

Fibonacci: $P(x) = 1/(2x^{1/\log_2 \phi}) \approx 1/(2x^{1.44})$, where $\phi = (1 + \sqrt{5})/2$ is the so-called *golden ratio*

Variable-Byte: $P(x) \approx \sqrt[7]{1/x^8}$

Zero- and Minimum-Redundancy Code

- **Zero-redundancy code.** If a code assigns codeword $C(x)$ such that $|C(x)| = I(x)$ bits for all x, then the code is optimal (in Shannon's sense) and is said to be a *zero-redundancy* code.

- But almost never $I(x)$ is not a whole number…

In the previous example for $L = [1,3,1,1,1,5,2,1,7,3,1,2,1,1,1,1]$, we had $\log_2(16/10) = 0.678$, but we cannot assign a codeword that is shorter than 1 bit!

- **Minimum-redundancy code.** Therefore, while zero-redundancy codes are impossible to achieve, we can compute a *minimum-redundancy* code that tries to minimise the overhead compared to the zero-redundancy code.
  (More about this in Module 4.)

Kraft-McMillan Inequality

Kraft, 1949 — McMillan, 1956

- **Q.** How short can codewords be so that the code can be prefix-free, thus, uniquely-decodable?
- We require every codeword length to be a whole number.
- If $P(x_i) = 1/2^{k_i}$ for some integer $k_i \geq 0$, then $I(x_i) = \log_2(1/(1/2^{k_i})) = k_i$ is a whole number and is the codeword length of x_i, i.e., $|C(x_i)| = k_i$.
- Since P is a distribution, it must hold:

$$\sum_{x_i} P(x_i) = \sum_{x_i} 2^{-k_i} = 1.$$

- Kraft noted that in such situations, it is possible to find a *prefix-free* code with codeword lengths equal to $\{k_i\}_i$.

x	$C(x)$
1	`0`
2	`10`
3	`110`
5	`1110`
7	`1111`

For this example, the sum is $1/2 + 1/4 + 1/8 + 2 \cdot 1/16 = 1$.

Kraft-McMillan Inequality

Kraft, 1949 — McMillan, 1956

- **Kraft-McMillan inequality.** Then it can be derived that

$$K = \sum_{x_i} P(x_i) = \sum_{x_i} 2^{-k_i} \leq 1$$

must hold for the prefix-free code to exist. In other words, we say that $\{k_i\}_i$ is a *valid* assignment of codeword lengths. In particular, if

1. $K < 1$: the code is valid but *not* optimal (at least one codeword can be shortened);
2. $K = 1$: the code is valid *and* optimal (no codeword can be shortened);
3. $K > 1$: the code is invalid (at least one codeword is shorter than what it should be).

- McMillan further observed that all is needed to specify a code is a *set of codeword lengths*: after provision is made for a set of codeword lengths satisfying the Kraft-McMillan inequality, then it is easy to assign prefix-free codewords and the specific codewords are *irrelevant.* (More about this in Module 4.)

- **Remark.** However, some assignments should be preferred over others to allow better encoding/decoding speed.

x	$C(x)$
1	0
2	10
3	110
5	1110
7	1111

Prefix-free and *lexicographic* codewords

x	$C(x)$
1	1
2	00
3	011
5	0101
7	0100

Other prefix-free but *non-lexicographic* codewords

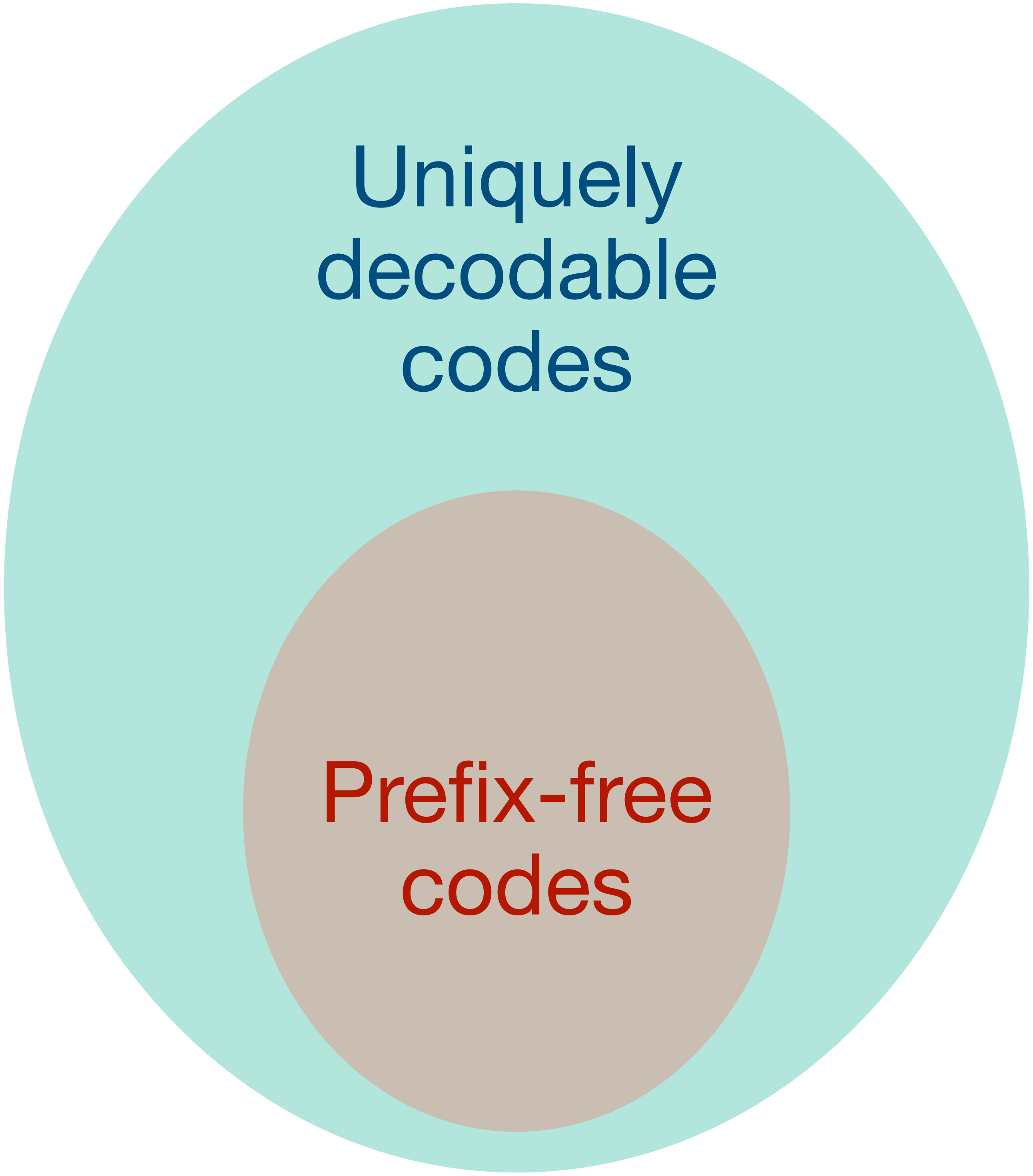
Kraft-McMillan Inequality

Kraft, 1949 — McMillan, 1956

- **Statement.** Given the symbols $\{x_i\}_{i=1}^m$ and a code C that encodes each symbol x_i with a codeword of length k_i, then C is *uniquely decodable* **if and only if**

$$\sum_{i=1}^{m} 2^{-k_i} \leq 1.$$

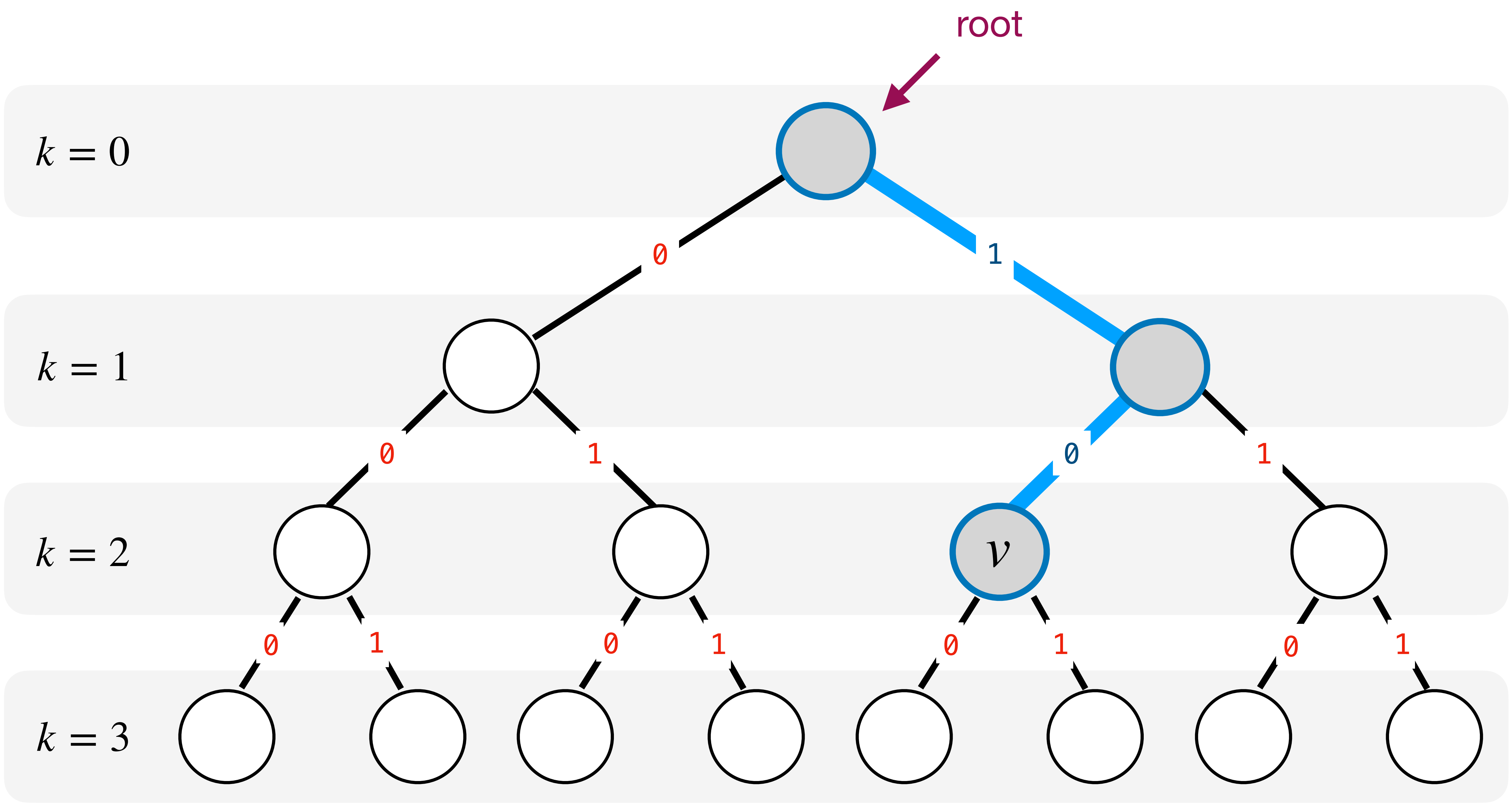
- **Remark.** The set containing all prefix-free codes is a subset of the one containing all the uniquely decodable codes.



Kraft-McMillan Inequality

- Let us consider a complete binary tree, where each left-leaning edge is labelled with a `0` and each right-leaning edge is labelled with a `1`.
- Then, it is convenient to think of a codeword of length k_i as the binary string spelled by a path from the root to a node at depth k_i.

The string spelled by the highlighted path from the root to the node v at depth $k = 2$ is "`10`".



Kraft-McMillan Inequality — Proof

Necessary condition for a prefix free code. If C is a prefix-free code, then $\sum_{i=1}^{m} 2^{-k_i} \leq 1$.

Proof. Each codeword length $k_i \leq k_{i+1}$, for $i = 1,\ldots,m-1$, corresponds to node v_i at depth k_i in a logical, complete, binary tree T with 2^{k_m} leaves (k_m is the longest codeword length). The subtree T_i rooted in v_i has $|T_i| = 2^{k_m-k_i}$ leaves. Since C is prefix-free, then $T_i \cap T_j = \varnothing$ for any two distinct subtrees T_i and T_j. Since the total number of leaves is 2^{k_m}, then $\left|\bigcup_{i=1}^{m} T_i\right| = \sum_{i=1}^{m} \left| T_i\right| = \sum_{i=1}^{m} 2^{k_m-k_i} \leq 2^{k_m}$. By dividing both sides of the inequality by 2^{k_m}, the result follows. ■

Kraft-McMillan Inequality — Proof

Sufficient condition for a prefix free code. If $k_1 \leq k_2 \leq \ldots \leq k_m$ is a set of codeword lengths such that $\sum_{i=1}^{m} 2^{-k_i} \leq 1$, then it is always possible to find a prefix-free code C with these codeword lengths.

Proof. Let us first try to build a prefix-free code C with the given codewords. We start by choosing one of the 2^{k_1} nodes at depth k_1 and by excluding all nodes in its subtree T_1. Then we select another node at depth k_2 among the ones that belong to $T \backslash T_1$. In general, at step j, we choose a node v_j at depth k_j among the ones that belong to $T \backslash \bigcup_{i=1}^{j-1} T_i$. We want to prove that this sequence of selections is always possible if $\{k_i\}_{i=1}^{m}$ satisfies the Kraft-McMillan inequality, that is, there are always some leaves left available to be covered with a subtree. This will prove that it is always possible to build a prefix-free code. Since we guarantee by construction that $T_{i'} \cap T_{i''} = \varnothing$ and $|T_i| = 2^{k_m - k_i}$, then the number of leaves left available after choosing v_j is equal to $2^{k_m} - \sum_{i=1}^{j} 2^{k_m - k_i} = 2^{k_m} \cdot \left(1 - \sum_{i=1}^{j} 2^{-k_i}\right)$. Therefore, even after the last node v_m is selected, we must have a non negative number of leaves left available because $\sum_{i=1}^{m} 2^{-k_i} \leq 1$ and so $2^{k_m} \cdot \left(1 - \sum_{i=1}^{m} 2^{-k_i}\right) \geq 0$. ■

Kraft-McMillan Inequality — Proof

- **Necessary condition for a uniquely-decodable code.** Any uniquely-decodable code satisfies the Kraft-McMillan inequality.

  Proof. Given the codeword lengths $k_1 \leq k_2 \leq \ldots \leq k_m$, let K be the sum $\sum_{i=1}^{m} 2^{-k_i}$. Then we define a new alphabet, whose symbols are concatenations of b symbols of our original alphabet $\{x_i\}_{i=1}^{m}$. The longest codeword in the new alphabet has length bk_m and $K^b = \left(\sum_{i=1}^{m} 2^{-k_i}\right)^b$ can be written as $K^b = \sum_{k=1}^{bk_m} q_k 2^{-k}$, where q_k is the number of codewords of length k. Since q_k must be at most 2^k, then $K^b \leq \sum_{k=1}^{bk_m} 2^k 2^{-k} = bk_m$. Taking the square root of order b of both sides we obtain $K \leq (bk_m)^{1/b}$ which must be satisfied for any $b \in \mathbb{N}$. Taking the limit for $b \to \infty$ it follows that

  $$\sum_{i=1}^{m} 2^{-k_i} = K \leq \lim_{b \to +\infty} (bk_m)^{1/b} = \lim_{b \to +\infty} e^{\frac{\ln(bk_m)}{b}} = 1. \blacksquare$$

- **Remark.** We do not need to prove the *sufficient* condition since we proved that if a set of codeword lengths satisfies the inequality, then we can always find a prefix-free code with those codeword lengths, and any prefix-free code is also uniquely-decodable.

Further Readings

- Section 2 of:
  G. E. P. and Rossano Venturini. 2020. *Techniques for Inverted Index Compression*. ACM Computing Surveys. 53, 6, Article 125 (November 2021), 36 pages. https://doi.org/10.1145/3415148

- Section 2.1-2.2 and Chapter 3 of:
  Alistair Moffat and Andrew Turpin. 2002. *Compression and coding algorithms*.
  Springer Science & Business Media, ISBN 978-1-4615-0935-6.

- Sections 1.1-1.5, 2.4, 2.19, 2.22, 2.23 of:
  David Salomon. 2007. *Variable-Length Codes for Data Compression*.
  Springer Science & Business Media, ISBN 978-1-84628-959-0.

- Sections 2.1-2.2-2.3 of:
  Gonzalo Navarro. 2016. *Compact Data Structures*. Cambridge University Press, ISBN 978-1-107-15238-0.