Lossless Coding

- Vector Huffman coding
- Unary and Golomb coding
- Run-length coding
- Arithmetic coding
- Universal source coding

Vector Huffman Coding

- Huffman coding very inefficient for *H(X)<<1* bit/symbol
- Remedy:
  - combine m successive symbols to a new “block-symbol”
  - Huffman code for block-symbols
  - Redundancy

$$\boxed{H(X) \leq R < H(X) + \frac{1}{m}}$$

  - Can also be used to exploit statistical dependencies between successive symbols
  - Disadvantage: exponentially growing alphabet size $\|\mathrm{A}_X\|^m$

Unary Coding

- "Geometric" source

$$\text{Alphabet } \mathrm{A}_X = \{0,1,\ldots\} = \mathbf{Z}_+ \qquad \text{PMF} \quad f_X(x) = 2^{-(x+1)}, \quad x \geq 0$$

- Optimal prefix code with redundancy $\rho=0$ is "unary" code ("comma code")

$$c_0 = "1" \quad c_1 = "01" \quad c_2 = "001" \quad c_3 = "0001" \quad \ldots$$

- Consider geometric source with faster decay

$$\text{PMF} \quad f_X(x) = (1-\beta)\beta^{\mathrm{x}}, \text{ with } 0 \leq \beta < \frac{1}{2}; \quad x \geq 0$$

- Unary code is still optimum prefix code (i.e., Huffman code), but not redundancy-free

Golomb Coding

- For geometric source with slower decay

$$\text{PMF} \quad f_X(x) = (1-\beta)\beta^x, \text{ with } \frac{1}{2} < \beta \le 1; \quad x \ge 0$$

- Idea: Express each x as

$$x = mx_q + x_r \quad \text{with} \quad x_q = \left\lfloor \frac{x}{m} \right\rfloor \quad \text{and} \quad x_r = x \bmod m$$

- Distribution of new random variables

$$f_{X_q}(x_q) = \sum_{i=0}^{m-1} f_X(mx_q + i) = \beta^{mx_q} \sum_{i=0}^{m-1} f_X(i)$$

$$f_{X_r}(x_r) = \frac{1-\beta}{1-\beta^m}\beta^{x_r} \quad \text{for} \quad 0 \le x_r < m$$

X_q and X_r statistically independent.

Golomb Coding

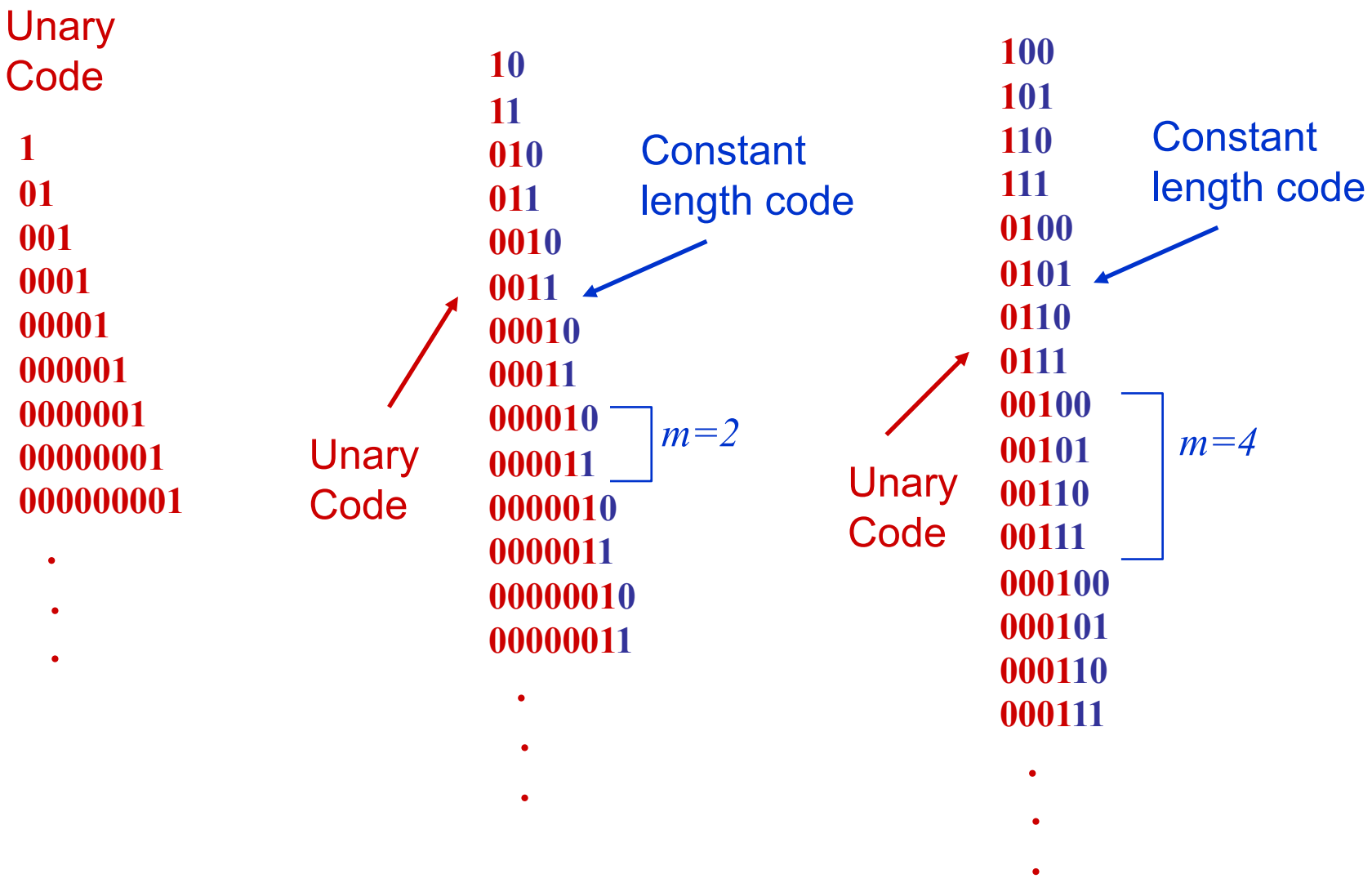
- Golomb coding
  - Choose integer divisor $\beta^m \approx \frac{1}{2}$
  - Encode x_q optimally by unary code
  - Encode x_r by a modified binary code, using code word lengths

$$k_a = \lceil \log_2 m \rceil \qquad k_b = \lfloor \log_2 m \rfloor$$

  - Concatenate bits for x_q and x_r

- In practice, $m = 2^k$ is often used, so x_r can be encoded by constant code word length $\log_2 m$

Example: Golomb Code



Golomb Parameter Estimation

- Expected value for geometric distribution

$$E[X] = \sum_{x=0}^{\infty} (1-\beta) x \beta^x = \frac{\beta}{1-\beta} \quad \rightarrow \quad \beta = \frac{E[X]}{1+E[X]}$$

- Approximation for $E[X] \gg 1$

Optimum performance of Golomb code

$$\beta^m = \frac{\left(E[X]\right)^m}{\left(1+E[X]\right)^m} \approx 1 - \frac{m}{E[X]} \approx \frac{1}{2}$$

$$m = 2^k \approx \frac{1}{2} E[X]$$

$$k = \max\left\{0, \left\lceil \log_2\left(\frac{1}{2}E[X]\right)\right\rceil\right\}$$

Reasonable setting, even if $E[X] \gg 1$ does not hold

Run-Length Coding

- For sources that emit "runs" of identical symbols
- Replace a sequence $\{x_n\}$ by a shorter sequence of symbol pairs $\{a_k, r_k\}$ such that

$$x_n = a_n \quad \text{for all } n : \sum_{j=1}^{k-1} r_j < n \leq \sum_{j=1}^{k} r_j$$

- Entropy coding (e.g., Huffman coding) of new symbol pairs $\{a_k, r_k\}$
- For binary sources, a_k can usually be omitted.

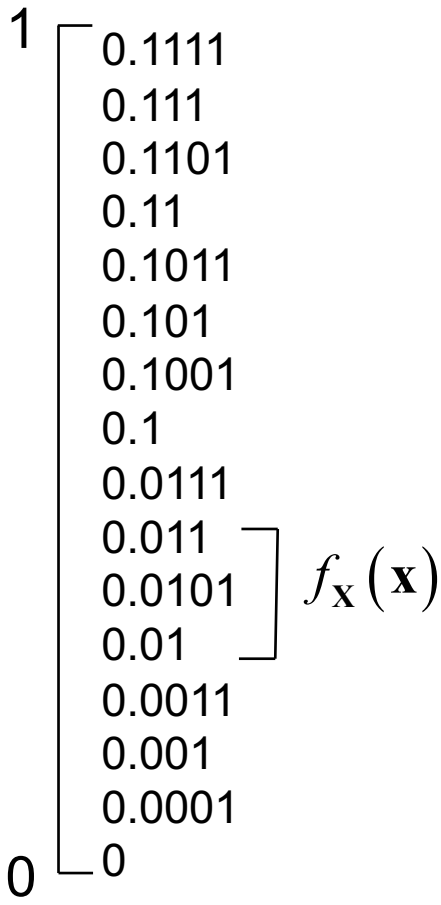
Arithmetic Coding

- Elias coding
- Arithmetic coding as finite-precision Elias coding
- Analysis of inefficiency due to finite precision

Elias Coding

- Entropy coding algorithm for sequences of symbols $\mathbf{x}$ with general (conditional) probabilities
- Representation of $\mathbf{x}$ by a subinterval of the unit interval $[0,1)$
- Width of the subinterval is approximately equal to the probability $f_{\mathbf{X}}(\mathbf{x})$
- Subinterval for $\mathbf{x}$ can be determined by recursive subdivision algorithm
- Represent $\mathbf{x}$ by shortest binary fraction in the subinterval
- Subinterval of width $f_{\mathbf{X}}(\mathbf{x})$ is guaranteed to contain one number that can be represented by L binary digits, with

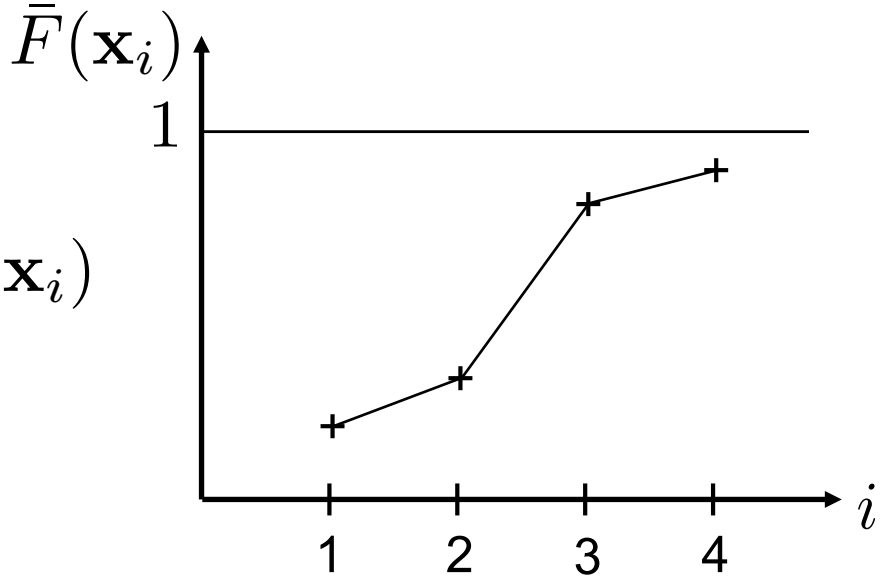
$$L \approx -\log_2 f_{\mathbf{X}}(\mathbf{x})$$



Elias Coding

- Define modified CDF

$$\bar{F}(\mathbf{x}_i) = \sum_{j \leq i} f(\mathbf{x}_j) - \frac{1}{2} f(\mathbf{x}_i)$$



- Binary representation of that value leads to codeword
- Truncate value such that it can be represented by *L* bits

$$\lfloor \bar{F} \rfloor_L = \sum_{i=1}^{L} b_i 2^{-i} \qquad b_i \in \{0, 1\}$$

Elias Coding

- $\bar{F}(\mathbf{x}_i)$ can be used for uniquely encoding / decoding any source sequence $\mathbf{x}_i$ as we have

$$\bar{F}(\mathbf{x}_k) \neq \bar{F}(\mathbf{x}_l) \Leftrightarrow k \neq l$$

- The truncation error is bounded by

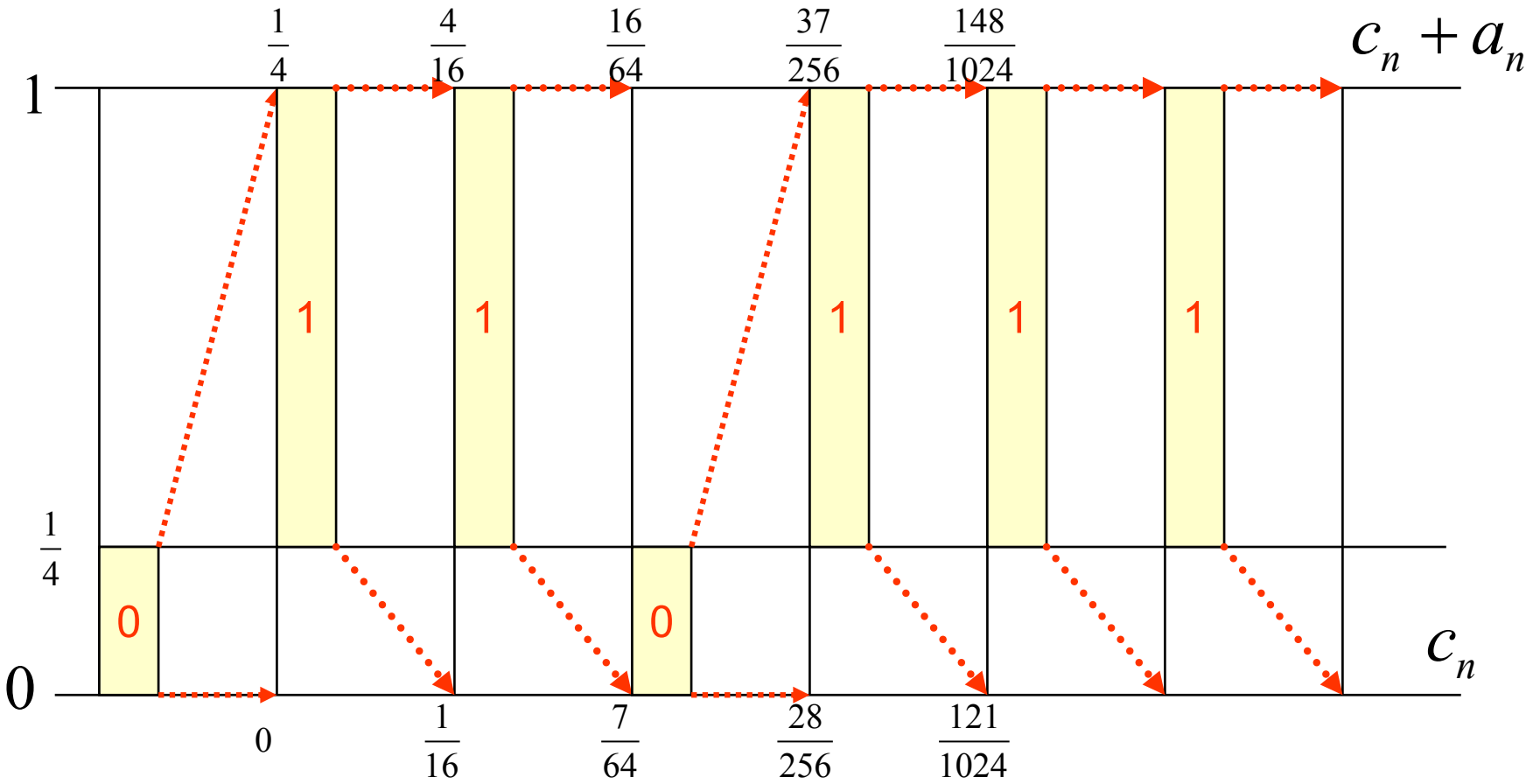
$$\bar{F}(\mathbf{x}_i) - \left\lfloor \bar{F}(\mathbf{x}_i) \right\rfloor_L < 2^{-L(\mathbf{x}_i)}$$

- For uniquely encoding / decoding, it is sufficient to have a binary string of length

$$L(\mathbf{x}_i) = \lceil -\log_2(f(\mathbf{x}_i)) \rceil + 1$$

Example: Elias Coding of Memoryless Binary Source

$$f_X(0) = \frac{1}{4} \qquad f_X(1) = \frac{3}{4}$$



Elias: Choose Binary Fraction in Subinterval

- Uniquely identify interval $[c_n, c_n + a_n)$ by a binary fraction

$$0.\underbrace{bbbbbb\ldots b}_{L_n \text{ bits}} \in [c_n, c_n + a_n)$$

- Truncation necessary to allow concatenation
- Length of bit string

$$L_n = \lceil -\log_2 a_n \rceil + 1$$

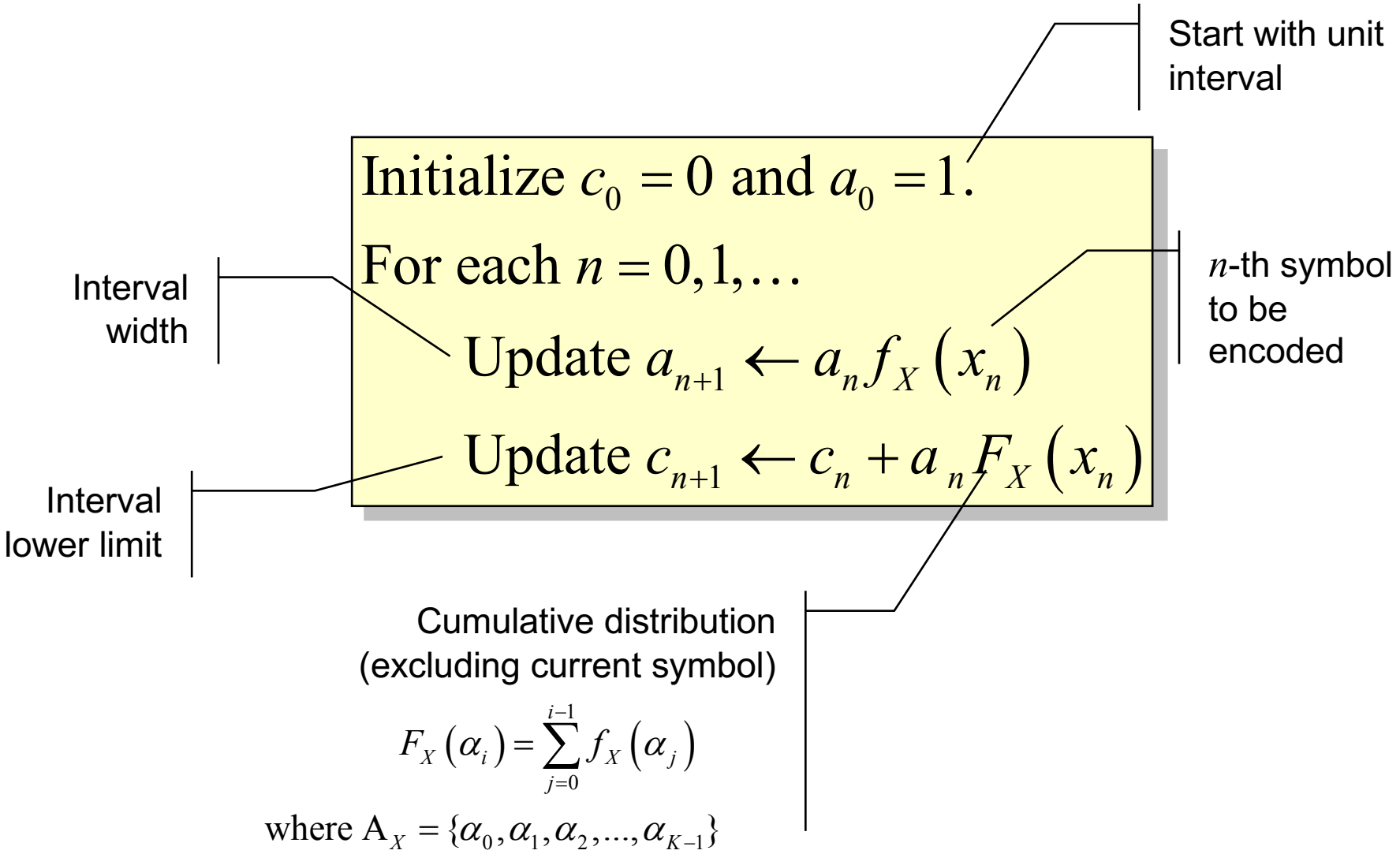
- Truncate such that lower boundary lies still in the interval

$$\hat{c}_n = 2^{-L_n} \left\lfloor 2^{L_n} c_n + 1 \right\rfloor > c_n$$

Successive Decoding of Elias Code

- Elias bit string can be decoded symbol by symbol, starting with the first symbol
- For each symbol $n=0,1,2,\ldots$
  - Calculate the intervals $[c_n, c_n+a_n)$ corresponding all possible x_n
  - Determine the interval for which $\hat{c}_n \in \left[c_n, c_n + a_n \right)$
  - Emit the corresponding x_n

Elias Coding for Memoryless Source



Elias Coding for Markov-p Source

- Calculate interval with conditional probabilities

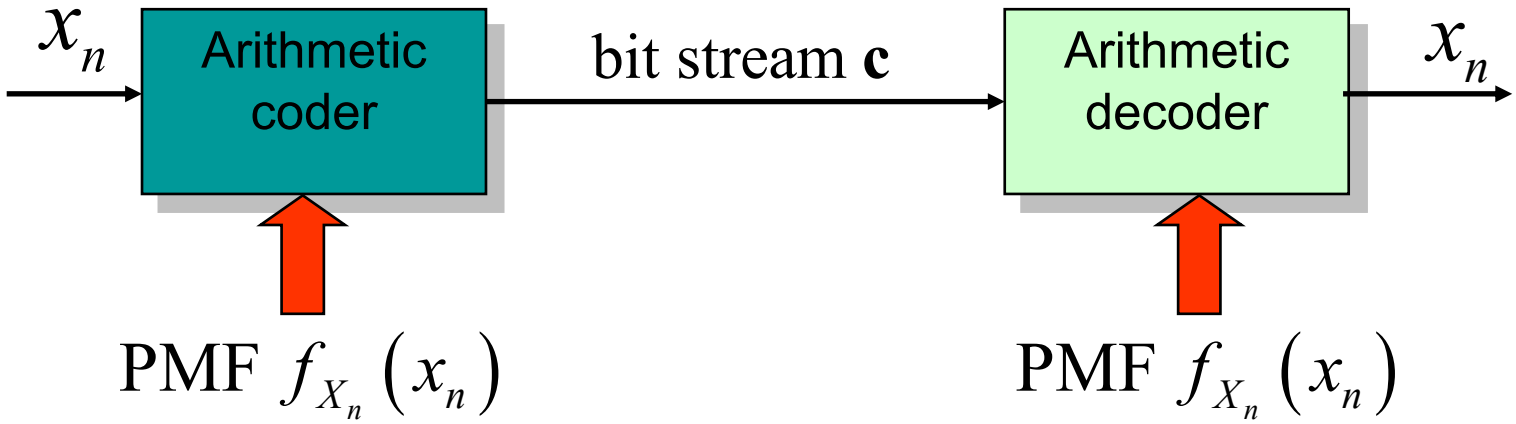
$$a_{n+1} \leftarrow a_n f_{X_n|X_{n-1},\ldots,X_{n-p}}(x_n|x_{n-1},\ldots,x_{n-p})$$

$$c_{n+1} \leftarrow c_n + a_n F_{X_n|X_{n-1},\ldots,X_{n-p}}(x_n|x_{n-1},\ldots,x_{n-p})$$

- Use cumulative conditional distribution (excluding current symbol)

Arithmetic Coding

- Elias coding not practical for long symbol strings: required arithmetic precision grows with string length
- Finite precision implementations: *"arithmetic coding"*
- Widely used in modern image and video compression algorithms: JPEG2000, H.264/AVC, H.265



Finite Precision for Arithmetic Coding

N-bit precision is permissible, if rounding down

Initialize $c_0 = 0$ and $a_0 = 1$.

For each $n = 0, 1, \ldots$

Update $a_{n+1} \leftarrow a_n f_X(x_n)$

Update $c_{n+1} \leftarrow c_n + a_n F_X(x_n)$

At most $N+P$ LSBs change, except for a possible carry affecting MSBs.
Each bit can be affected by at most one carry.

P-bit approximation of probabilities

Finite Precision for Arithmetic Coding

- Output code string

$$c_n = .\underbrace{xxxxxxxxx\ldots xxx}_{\text{MSBs that will no longer change}}\ \underbrace{01111111\ldots 1}_{r_n+1 \text{ bits that might be affected by a carry}}\ \underbrace{ccccccccc\ldots c}_{N+P \text{ LSBs that represent lower interval boundary for further computation}}$$

Send	Represent by state variable r_n	Represent by state variable C_n

- Maximum value of r_n can be limited by bit stuffing

Inefficiency due to Interval Rounding

- Recall: Subinterval of width $f_{\mathbf{X}}(\mathbf{x})$ is guaranteed to contain one number that can be represented by L_n binary digits, with

$$L_n = \lceil -\log_2 a_n \rceil + 1$$

- Hence, rounding one interval value a_{n+1} to N-bit precision increases bit string by

$$\log_2 \frac{a_n f_X(x_n)}{a_{n+1}} < \log_2 \frac{2^{N-1}+1}{2^{N-1}} = \log_2\left(1+2^{1-N}\right) \approx \frac{1}{\ln 2} 2^{1-N}$$

Limited Precision Probabilities

- Efficiency loss:

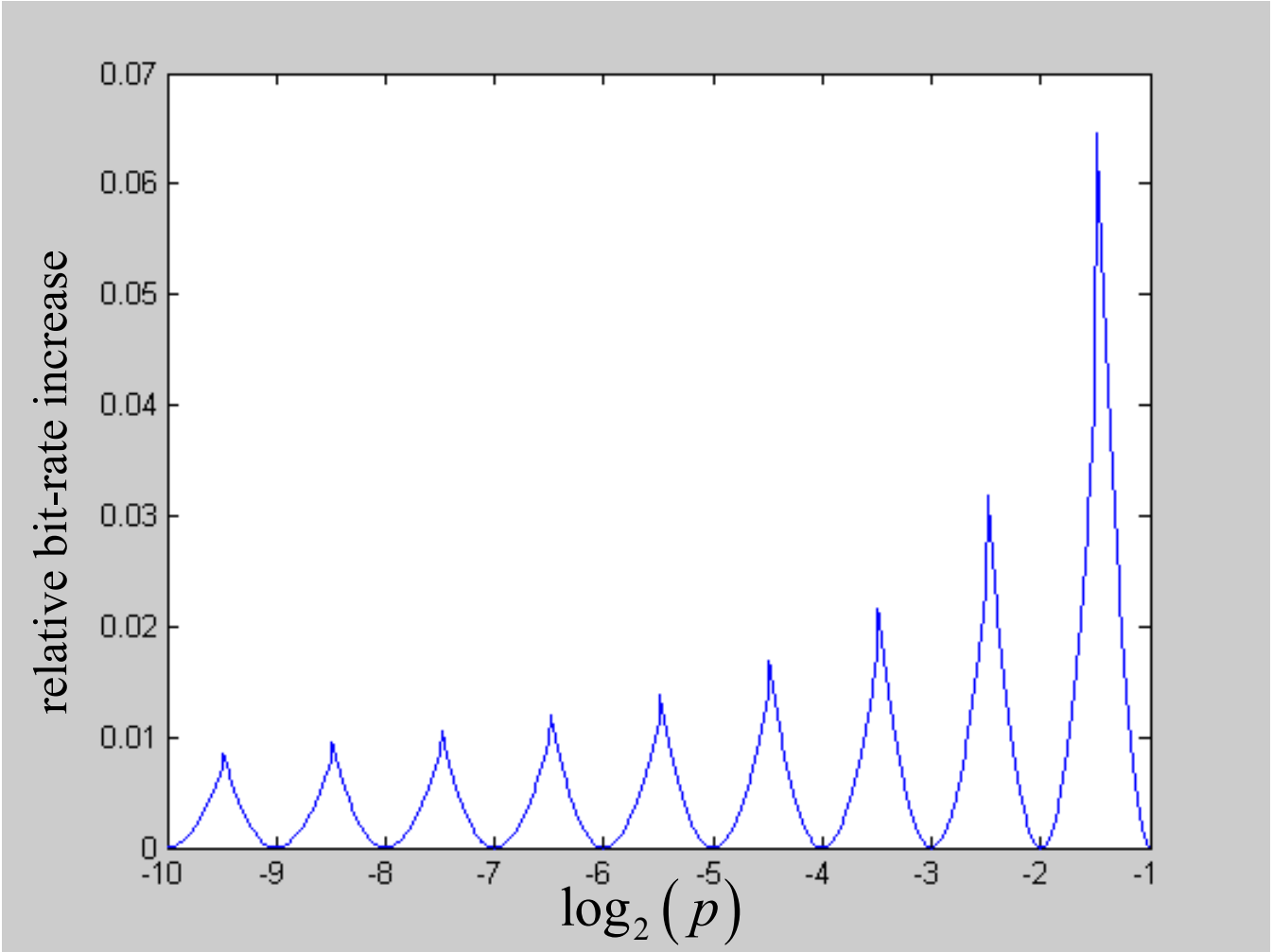
$$E\left[-\log_2\left(f_X(x)\left(1+\frac{\Delta f_X(x)}{f_X(x)}\right)\right)\right] = \underbrace{E\left[-\log_2\left(f_X(x)\right)\right]}_{=H(X)} - \underbrace{E\left[\log_2\left(1+\frac{\Delta f_X(x)}{f_X(x)}\right)\right]}_{\approx \frac{1}{\ln 2} E\left[\frac{\Delta f_X(x)}{f_X(x)}\right]}$$

$$E\left[\frac{\Delta f_X(x)}{f_X(x)}\right] = \sum_x f_X(x) \cdot \frac{\Delta f_X(x)}{f_X(x)} = \sum_x \Delta f_X(x) = 0$$

provided that rounded probabilities still add to 1

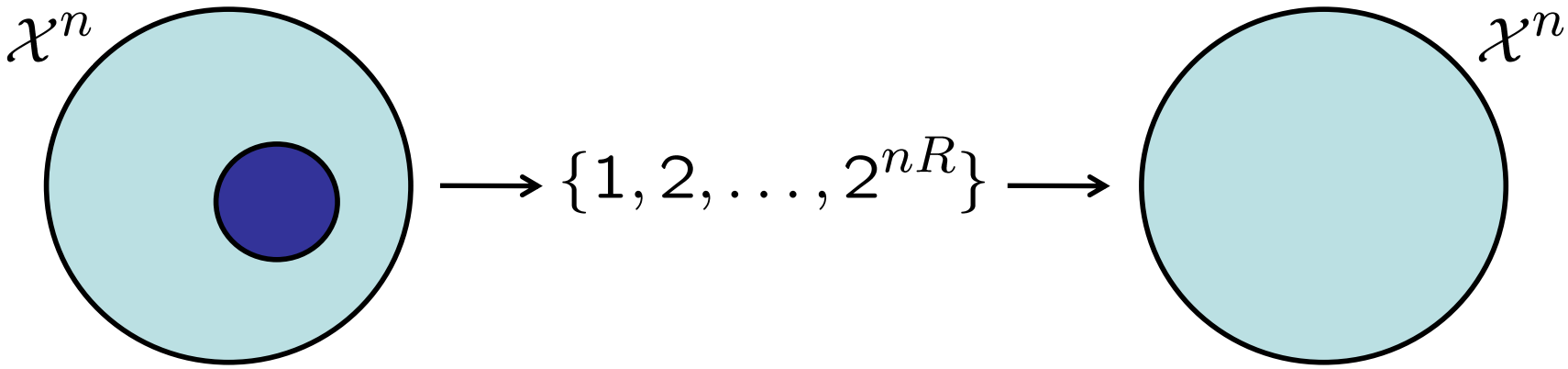
Example: Inefficiency for Binary Source

Representing smaller probability by 2^{-P}



Universal Source Coding

- Let the encoder map n symbols of the source X to a rate R code sequence
- Let the decoder be able to recover the n source symbols



- A source code is called universal if the mappings do not depend on the distribution

Lempel-Ziv Coding

- Lempel-Ziv algorithm:
  - Replaces strings of characters by single codes
  - Adds every new string of characters to a table of strings
  - Compression occurs when the table of strings contains long strings of characters that are also frequently used
- Universal codes need a longer block length to obtain the same performance as a code designed specifically for the probability distribution

- A distribution specific code is best if one knows the distribution of the source