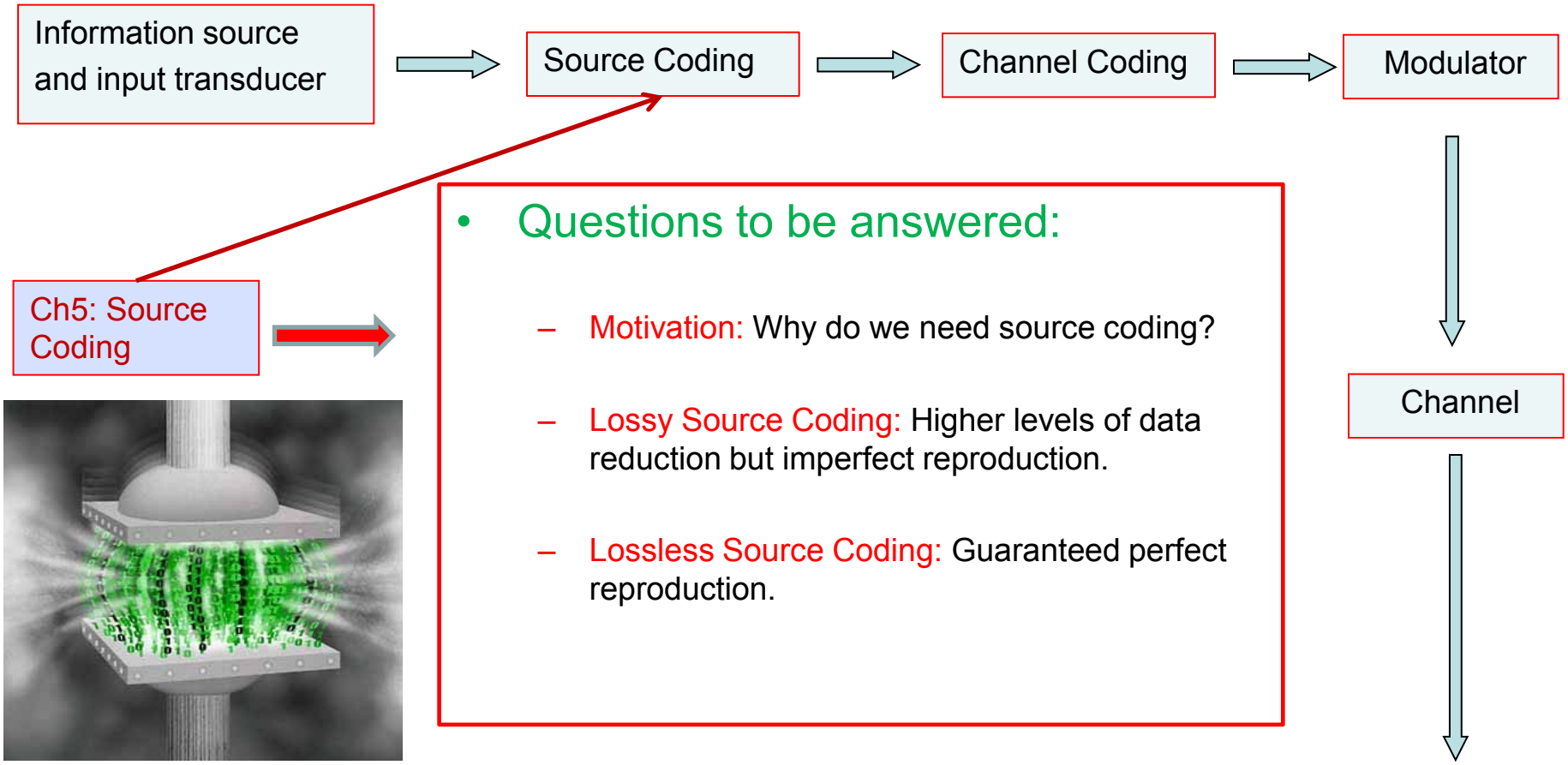
Ch5: Source Coding



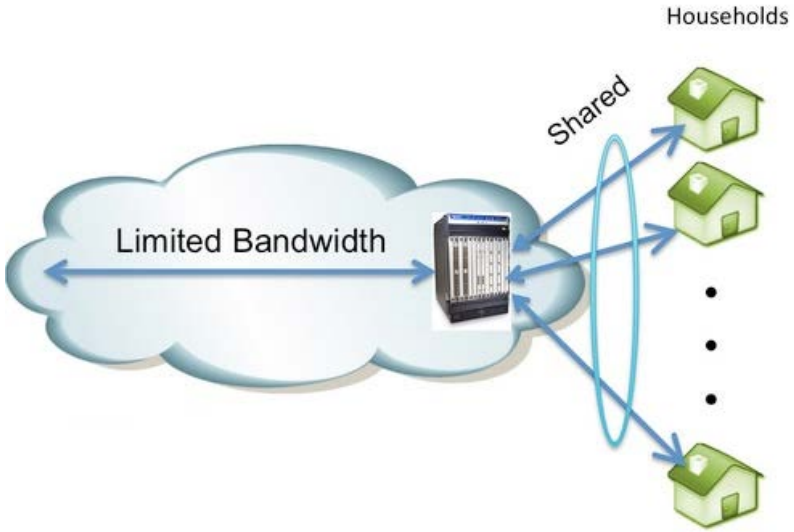
Ch5: Source Coding

- **Motivation**

- Lossy Source Coding
  - PCM
  - Transform Coding

- Lossless Source Coding
  - Entropy Coding
  - Hoffman Coding
  - LZW Coding

Why We Need Source Coding

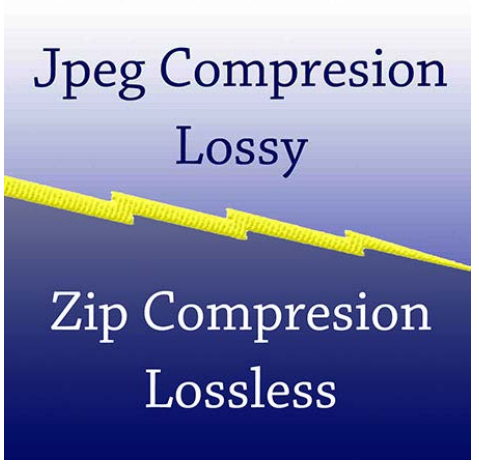


Signal		Uncompressed Rate	Common Rate
Music		4.32 Mbit/s (CD audio)	128 Kbit/s (MP3)
Voice		64 Kbit/s (AM radio)	4.8 Kbit/s (cellphone CELP)
Photos		14 MB (raw)	300 KB (JPEG)
Video		170 Mbit/s (PAL)	700 Kbit/s (DivX)

1 byte = 8 bit

Types of Source Coding

- Lossy Source Coding:
  - **Lossy:** Provides higher levels of data reduction but results in a less than perfect reproduction of the original data.
  - Depends on characteristics of the data
  - e.g. PCM, DPCM, ADPCM



- Lossless Source Coding
  - **lossless**
  - e.g. Entropy, Huffman, LZW

Quantization step → data lost!

need to specify min $(x-\hat{x})^2$ ← distortion!

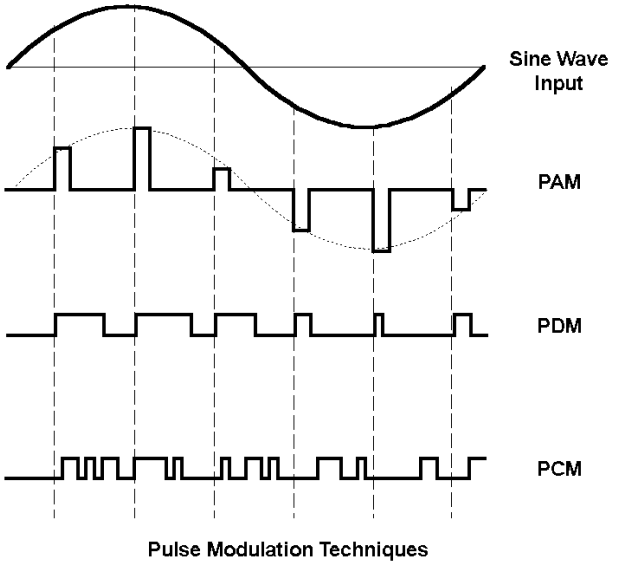
min k → rate distortion tradeoff

$$compression\ ratio = \frac{B_0}{B_1}$$

B_0 – number of bits before compression
B_1 – number of bits after compression

Ch5: Source Coding

- Motivation
- Lossy Source Coding
  - **PCM**
  - **Transform Coding**
- Lossless Source Coding
  - Entropy Coding
  - Hoffman Coding
  - LZW Coding



Pulse Modulation Techniques

Analog-to-Digital Conversion: Sampling and Quantization

- **Sampling**
  - makes signals discrete in time
  - bandlimited signals can be sampled without introducing distortion

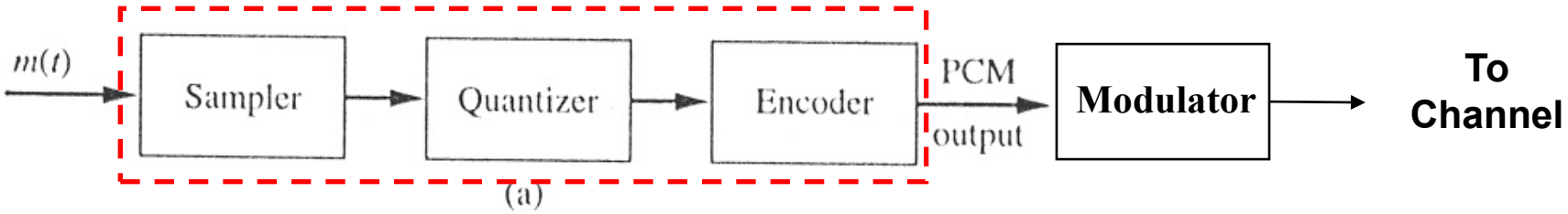
- **Quantization**
  - makes signal discrete in amplitude
  - introduce some distortion

Pulse Code Modulation (PCM)

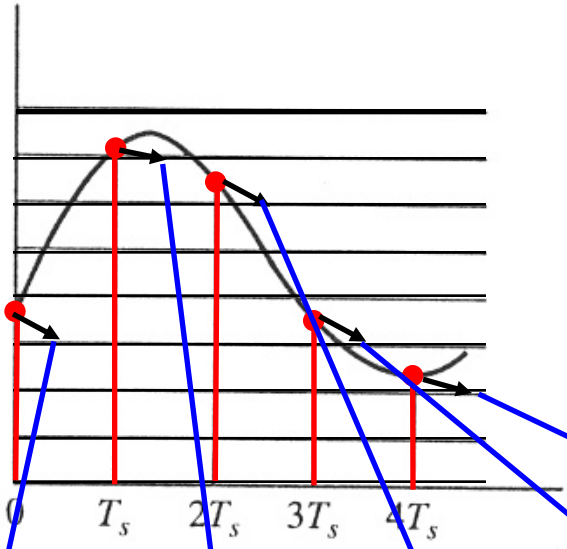
- **PCM** is a modulation scheme in which the **amplitude of the sampled signal is converted to a set of quantized levels**
- **Each quantized level is translated into a binary code**
  - For L (= 2^n) quantized levels, number of bit per code (or per sample) is n (=$\log_2 L$)
- **The binary code is then converted into sequential string of pulses for transmission**

Analog-to-Digital Conversion

Analog-to-Digital Converter (ADC)

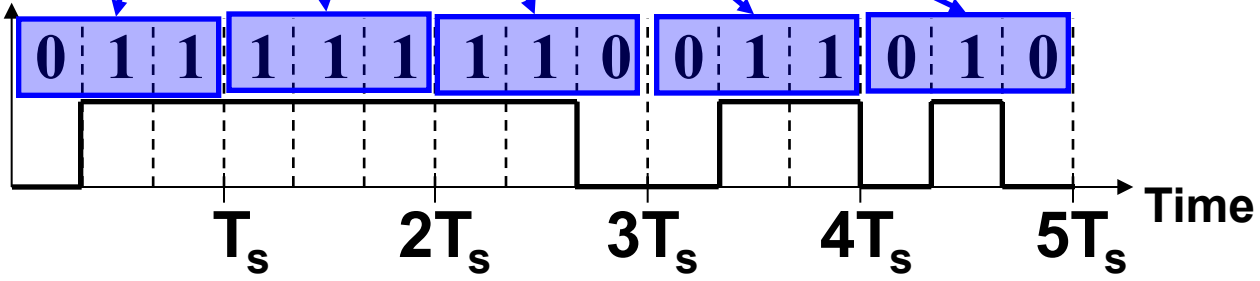


Quantization level number	Encoded output
7	111
6	110
5	101
4	100
3	011
2	010
1	001
0	000

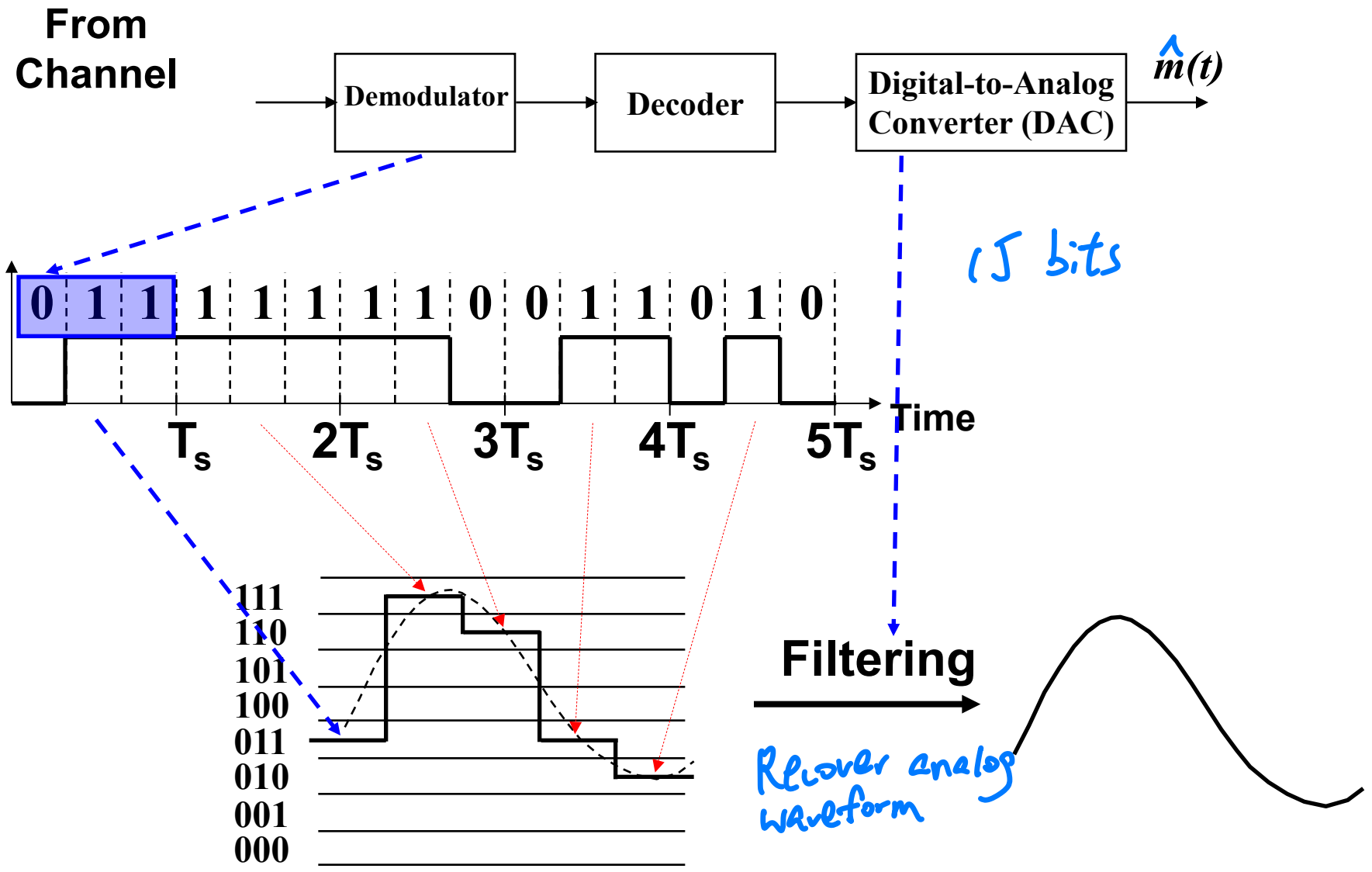


- **Sampling**

3 bits to represent one sample!



Digital-to-Analog Conversion



Pulse Code Modulation: Bandwidth

- For L ($=2^n$) quantized levels, n ($=\log_2 L$) binary pulses must be transmitted for each sample point of the message signal.

- If the message bandwidth is B and the sampling rate is f_s ($\geq 2B$), then nf_s binary pulses must be transmitted per second as each sample point will be translated into a n-bit code.

in one second!

Pulse Code Modulation: Modem

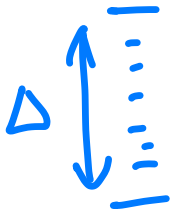
Applications: Telephone System

Voice bandwidth ~ 4kHz → minimum sampling rate: 8kHz

→ 8000 samples per second

8-bit PCM is used → 8 bits per sample → 8x8000 = 64kbits per second

For modem application, only 7 bits are for data, the other is an overhead bit → 56kbit/s



Quantization Noise

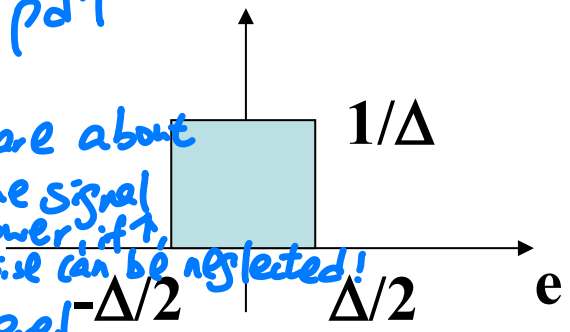
- **Quantization noise** is created in PCM because of assigning continuous sample values to the nearest **quantization level**

- Let e= **Quantization error** $(x(t) - x_q(t))$

  $\Delta =$ **Quantization step**

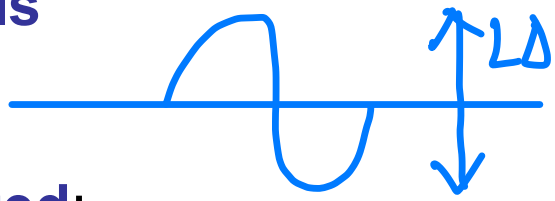
- Assume **errors** are **uniformly distributed** over a step interval then

$$\overline{e}^2 = E[e^2] = \int_{-\Delta/2}^{\Delta/2} e^2 \frac{1}{\Delta} de = \frac{\Delta^2}{12}$$



Quantization Noise

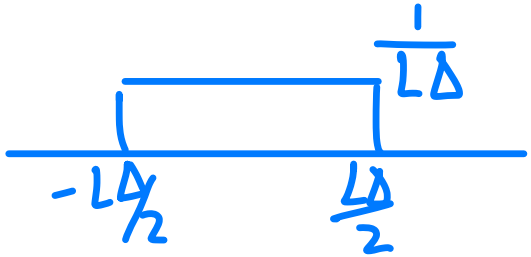
- Let L = **Number of quantization levels**



- Assuming a **sine wave** is to be **quantized**:
  - Then, the **sine wave amplitude** = $\frac{L\Delta}{2}$
  - **Average power** = $\frac{(L\Delta/2)^2}{2} = \frac{L^2\Delta^2}{8}$

$$\int_{2\pi} \left(\frac{L\Delta}{2}\right)^2 \sin^2 \omega t = \frac{L^2\Delta^2}{8}$$

- Alternatively, assuming a random uniformly distributed signal:
  - **Average power** = $\frac{L^2\Delta^2}{12}$



Quantization Noise

- Hence, **Signal-to-quantization noise ratio**

$$SNR_Q = \frac{L^2\Delta^2/8}{\Delta^2/12} = \frac{3}{2}L^2$$

or

$$SNR_Q = \frac{L^2\Delta^2/12}{\Delta^2/12} = L^2$$

- Next recall that the samples are **transmitted by binary words**

- Since there are **L levels**, it follows that there are ⟹ $N = \log_2(L)$ bits/word

Quantization Noise

- That is, $L = 2^N$ N = # of bits Consider Rate/Distortion!
- Thus, implying $SNR_Q = \frac{3}{2}2^{2N}$
- Or in dB form

Linear relationship!

$$SNR_Q = 6N + 1.761 \qquad (dB)$$

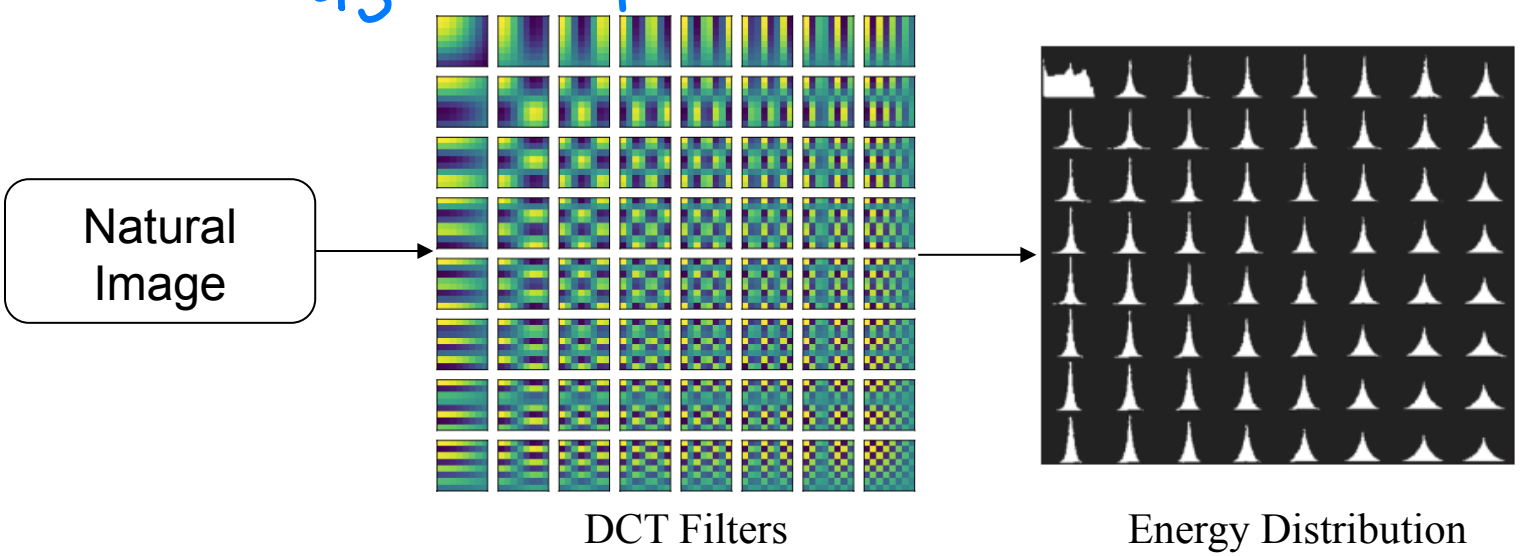
tradeoffs!

i.e., we gain 6 dB in SNR for every bit added to the quantizer word length.

- Can you figure out a better quantization method?
- DPCM, ADPCM, etc. can use correlation

Transform Coding

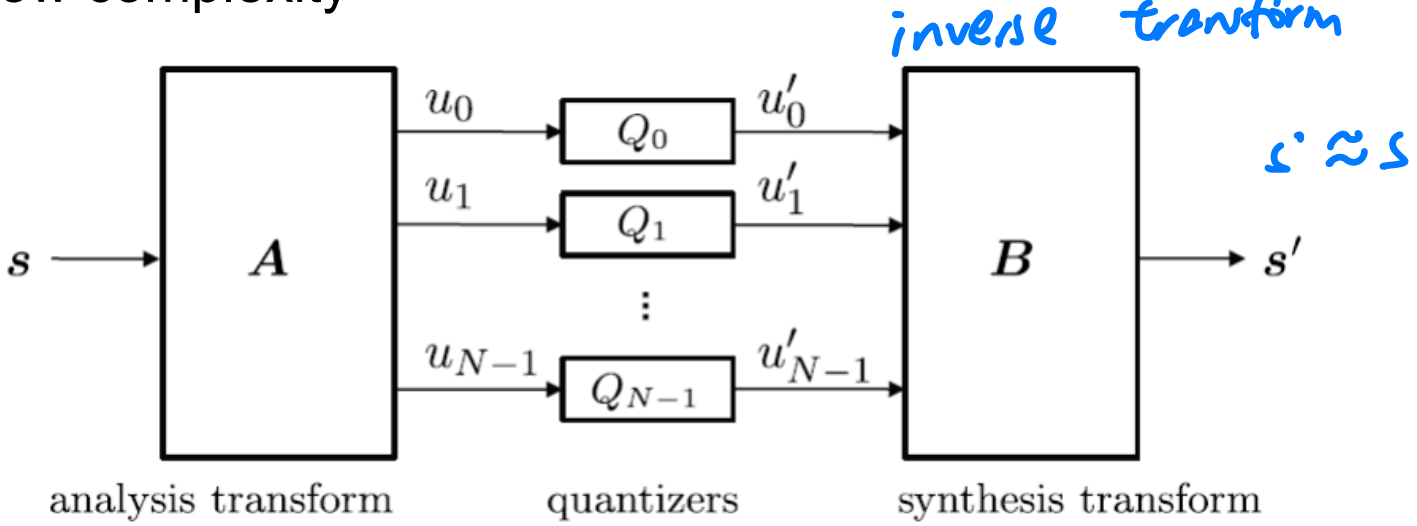
- **Transform coding**: transform data into another compact domain so that the data becomes uncorrelated for follow-up processing, converting data into a space where coding is easier.
- Motivation: Decorrelation and energy concentration.
- Example: Discrete Cosine Transform (DCT) in JPEG



Transform Coding

lossy!

- **Advantages**:
- Thanks to the energy compaction in transform space, quantization in the transform space is more effective.
- Exploit the statistical correlations and dependencies of a source at low complexity



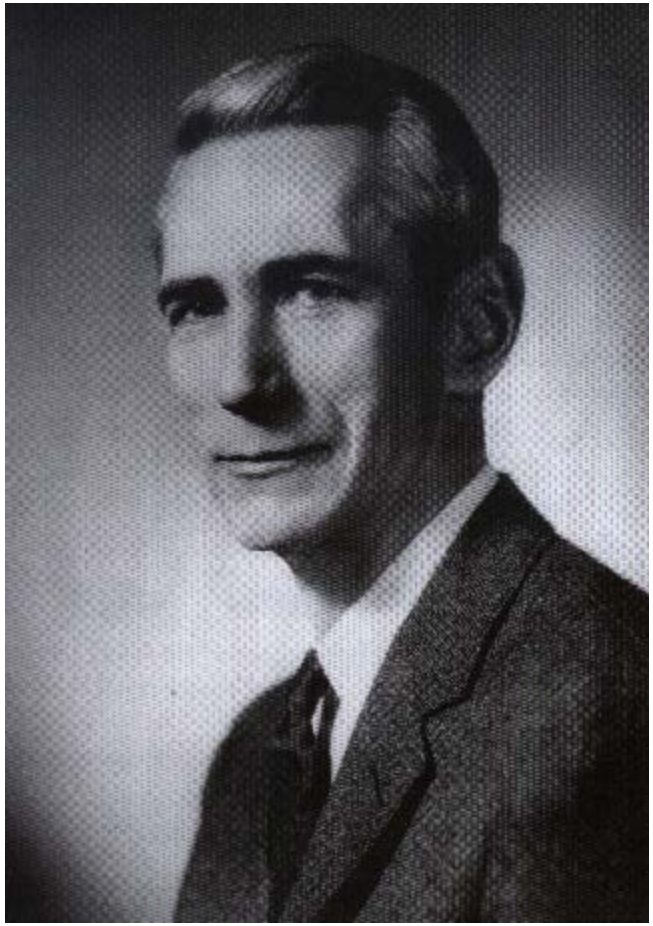
Example of a transform coding system
s in data space, u in transform space

Ch5: Source Coding

- Motivation
- Lossy Source Coding
  - PCM
  - Transform Coding
- **Lossless Source Coding**
  - Entropy and Codes
  - Shannon and Fano's Coding
  - Huffman Coding
  - Arithmetic Coding
  - LZW Coding

increase the efficiency for digital signal!

Father of Information Theory



Claude E. Shannon
1916 - 2001

The foundation of our Information Age is the transformation of speech, audio, images and video into digital content, and the man who started the digital revolution was **Claude Shannon**, who died on February 24, 2001 at the age of 84.

http://www.itsoc.org

Claude Shannon

- Shannon single-handedly started **Information Theory** by publishing the paper "A Mathematical Theory of Communication" in 1948 while working in AT&T Bell Labs.
- Information theory described the measurement of information by binary digits - the fundamental basis of today's telecommunications
- The term **bit** was coined by Shannon!
- His early projects included
  - a computer system that predicts a coin toss
  - a mechanical mouse that could learn to run a maze
  - chess playing machines

Entropy

Minimize the number of bits !

- **Entropy:** is a measure of disorder or, more precisely unpredictability or information content.

- In information theory, **entropy** is a measure of the uncertainty associated with a random variable .

- Shannon denoted the entropy *H* of a discrete random variable with possible values as: $\{x_1, \ldots, x_n\}$

$$H(X) = -\sum_{i=1}^{n} p(x_i)\log_2(p(x_i)) = \sum_{i=1}^{n} p(x_i)[-\log_2 p(x_i)]$$

$$= E[-\log_2 p(x_i)]$$

where

PMF : $p(x_i) = P[X = x_i]$.

e.g. $x_i = 0, 1$

$x_i = A, B$

x_i = Apple, Orange

Only know the distribution

Entropy

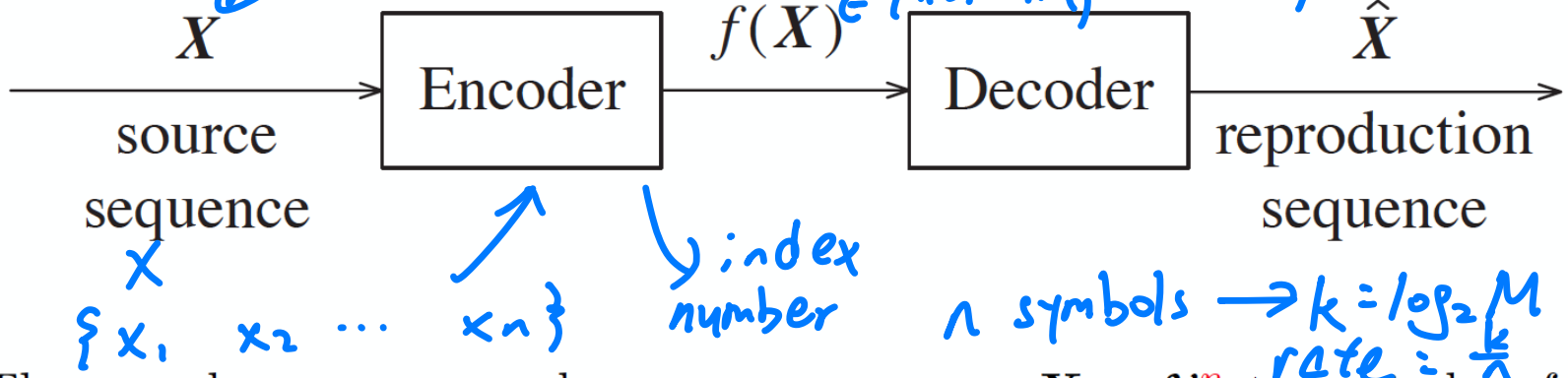
- **Example:** Entropy calculation for a two symbol alphabet $S=\{x_1, x_2\}$.
  - Case 1: $p(x_1)=p(x_2)=0.5$
  - Case 2: $p(x_1)=0.8,\ p(x_2)=0.2$

- Solution: $H(X)=-\sum_{i=1}^{n} p(x_i)\log_2(p(x_i))$
  - Case 1: $H(X)=-0.5\log_2 0.5-0.5\log_2 0.5=1$
  - Case 2: $H(X)=-0.8\log_2 0.8-0.2\log_2 0.2=0.7219$

  Understanding:
  - *It requires one bit per symbol on the average to represent the data for case 1.*
  - *It requires 0.7219 bit per symbol on the average to represent the data for case 2.*

- How can we achieve this?

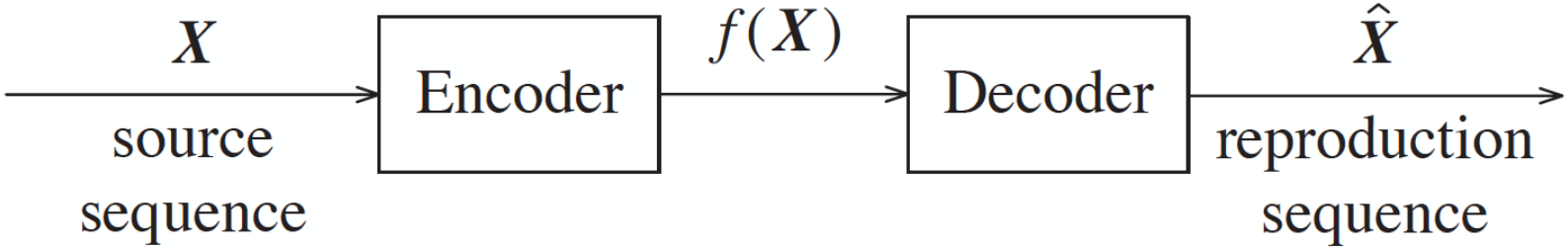
A Source Code



- The encoder maps a random source sequence $\mathbf{X} \in \mathcal{X}^n$ to an index $f(\mathbf{X})$ in an index set

$$\mathcal{I} = \{1, 2, \cdots, M\}.$$

- Such a code is called a block code with n being the block length of the code.
- The encoder sends $f(\mathbf{X})$ to the decoder through a noiseless channel.
- Based on the index, the decoder outputs $\hat{\mathbf{X}}$ as an estimate on $\mathbf{X}$.



- The encoder is specified by the mapping

$$f : \mathcal{X}^n \to \mathcal{I} = \{1, 2, \cdots, M\}.$$

- The rate of the code is given by $R = n^{-1} \log_2 M$ in bits per source symbol, where M is the size of the index set and n is the block length.
- If $M = |\mathcal{X}^n| = |\mathcal{X}|^n$, the rate of the code is

$$\frac{1}{n} \log_2 M = \frac{1}{n} \log_2 |\mathcal{X}|^n = \log_2 |\mathcal{X}|.$$

- Typically, $R < \log |\mathcal{X}|$ for data compression.
- An error occurs if $\hat{\mathbf{X}} \neq \mathbf{X}$, and $P_e = \Pr\{\hat{\mathbf{X}} \neq \mathbf{X}\}$ is called the error probability.

The Source Coding Theorem

Nothing is perfect!

Direct Part: For arbitrarily small P_e, there exists a block code whose coding rate is arbitrarily close to $H(X)$ when n is sufficiently large. large block!

- This part says that reliable communication can be achieved if the coding rate is at least $H(X)$.

Rate $\geq H(X)$

how small R can be!

Converse: For any block code with block length n and coding rate less than $H(X) - \zeta$, where $\zeta > 0$ does not change with n, then $P_e \to 1$ as $n \to \infty$.

↑ small

Reconstruction is impossible :<

- This part says that it is impossible to achieve reliable communication if the coding rate is less than $H(X)$.

$R < H(X) \Rightarrow$ reliable communication is impossible

☆ Distribution
→ Can calculate entropy
→ Can know best solution
→ how? have very good algorithm!

Entropy and Coding

Ideal case! Very ideal!!!

- Entropy is a ***lower bound*** on the average number of bits needed to represent the symbols (**the data compression limit**).
- Aspire to achieve the entropy for a given alphabet,

how good?

$$BPS(\text{bits per symbol}) = \frac{|\text{Encoded Message}|}{|\text{Original Message}|} \rightarrow \text{Entropy}$$

- A code achieving the entropy limit is optimal.
- A binary code encodes each character as a binary string or codeword.
- The average rate achieved by a code is $R = \sum_{i=1}^{n} p(x_i) l(x_i)$

  where $l(x_i)$ is the length of the *i*th symbol.

number of bits!

Entropy

$H(x) = \sum p(x_i)(-\log_2 p(x_i))$

$= E[-\log_2(p_{x_i})]$

What is the best that we can do?

Want $l_i = \log_2 \frac{1}{p_{x_i}}$

↓

yr code is optimal (Entropy)

Rate (Average rate of code)

$R = \sum \underline{p(x_i)} \underline{\underline{l_i}} \geq$ Entropy

l_i = length of bits of ith symbol

☆ If $l_i = -\log_2 p(x_i) \Rightarrow$ Rate = Entropy

$\simeq \log_2 \frac{1}{p(x_i)}$ If $p(x_i)$ small, l_i should be large

↑ difficulty: to be integer

If each symbol is uniform distributed!

Need MORE length of codes!

(FYI) Kraft's Inequality for Codes

- A code with codeword lengths l_1, ..., l_N, exists if and only if

$$\sum_{i=1}^{N} 2^{-l_i} \le 1$$

不能有 negative length
infinite length

- If you want to patent a new code and it does not satisfy Kraft's inequalities, the patent office will reject it.

- **Proof**: Let $l_{max} = \max\{l_1, ..., l_N\}$. Expand the tree so that all branches have depth l_{max}. A codeword at depth l_i has $2^{l_{max}-l_i}$ leaves underneath itself at depth l_{max}. The sets of leaves under codewords are disjoint. The total number of leaves under codewords are less than or equal to $2^{l_{max}}$. Thus $\sum_{i=1}^{N} 2^{l_{max}-l_i} \le 2^{l_{max}}$ or, equivalently, $\sum_{i=1}^{N} 2^{-l_i} \le 1$

- One consequence of Kraft's ineq is that $R \ge H(X)$.

Fixed-Length vs. Variable-Length Codes

- Code types:
  - Fixed-length codes: all codewords have the same length (number of bits)
  - Variable-length codes (prefix codes): different codewords may have different number of bits
- Consider a file with 100,000 character data file that we wish to store. The file contains only 6 characters with different frequencies.

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
a fixed-length	000	001	010	011	100	101
a variable-length	0	101	100	111	1101	1100

most frequently !

- The fixed-length code requires 300,000 bits to store the file.
- The variable-length code uses only 224,000 bits. (25% reduction !!!)

Simple coding scheme

Shannon-Fano's Coding

May not be optimal

- Shannon and Fano proposed a coding scheme independently around 1944.
- Order the symbols according to their probability (frequency) and split this list into two groups with roughly equal probability. Prefix all the codes of the symbols in the first group with 0 and in the second group with 1.
- Repeat recursively this procedure until you get lists with a single symbol.
- Very simple but it can be improved. It is not optimal. The method is greedy and at each step you don't know if you are doing the right thing.

Shannon-Fano's Coding

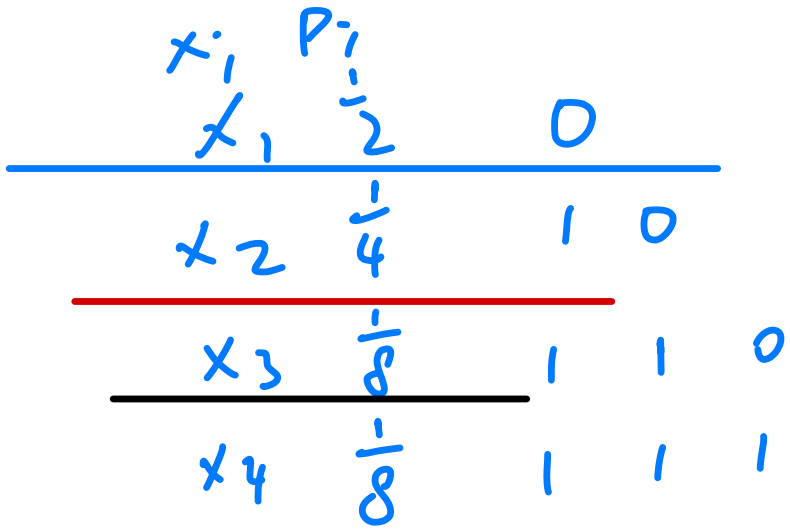
Symbol	Frequency	Code
a	38416	00
b	32761	01
c	26896	100
d	14400	101
e	11881	1100
f	6724	1101
g	4225	1110
h	2705	1111

a)

Symbol	Frequency	Code
a	34225	000
b	28224	001
c	27889	01
d	16900	100
e	14161	101
f	4624	110
g	2025	1110
h	324	1111

b)

Table 1.1: Examples of Shannon-Fano codes for 8 symbols. a) a codebook with codes of average length 2.67. The entropy of this distribution is 2.59 bits/symbol. Discrepancy is only 0.08 bits/symbol. b) an example of a Shannon-Fano codebook for 8 symbols exhibiting the problem resulting from greedy cutting. The average code length is 2.8, while the entropy of this distribution is 2.5 bits/symbol. Here, discrepancy is 0.3 bits/symbol. This is much worse than the discrepancy of the codes shown in a).



Check entropy is optimal

Optimal algorithm

Must be optimal!
Immediately decodable!

Huffman Coding

by some course project → student do this!!!

- Instead of going from top to bottom, Huffman had the idea in 1952 of attacking the problem from bottom up. It is optimal.
- Each symbol is assigned a variable-length code, depending on its frequency. The higher its frequency, the shorter the codeword.
- Number of bits for each codeword is an integer number.

- Codewords are generated by building a Huffman Tree to determine Huffman code, it is useful to construct a binary tree
- Leaves are characters to be encoded
- Nodes carry occurrence probabilities of the characters belonging to the subtree

- Example: How does a Huffman code look like for symbols with statistical symbol occurrence probabilities:

  P(A) = 1/8, P(B) = 1/8, P(C) = 1/4, P(D) = 1/2?

Huffman Encoding (Example)

Step 1 : Sort all Symbols according to their probabilities (left to right) from Smallest to largest

these are the leaves of the Huffman tree

P(D)=0.5

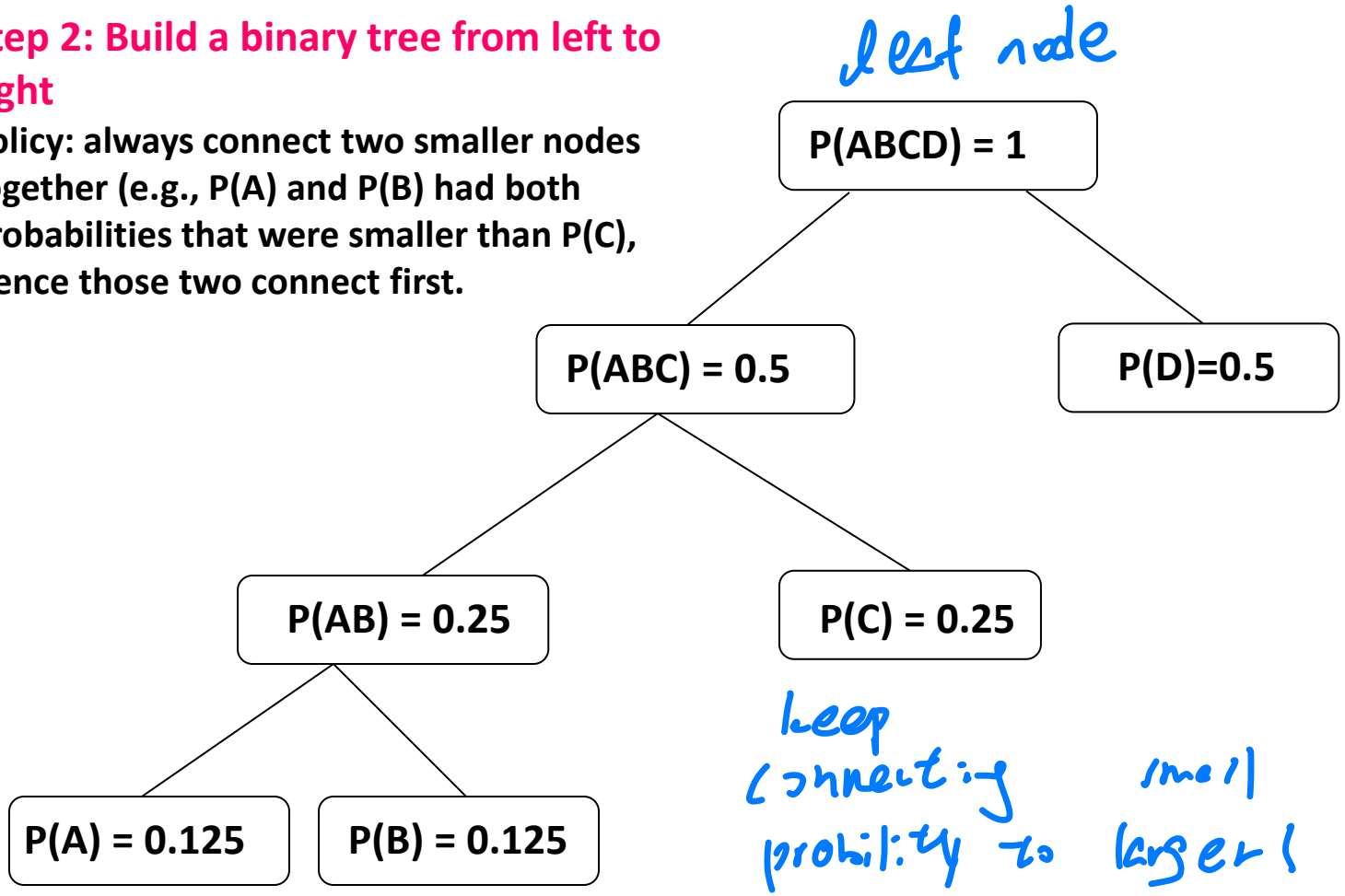
P(C) = 0.25

P(A) = 0.125 **P(B) = 0.125**

Huffman Encoding (Example)

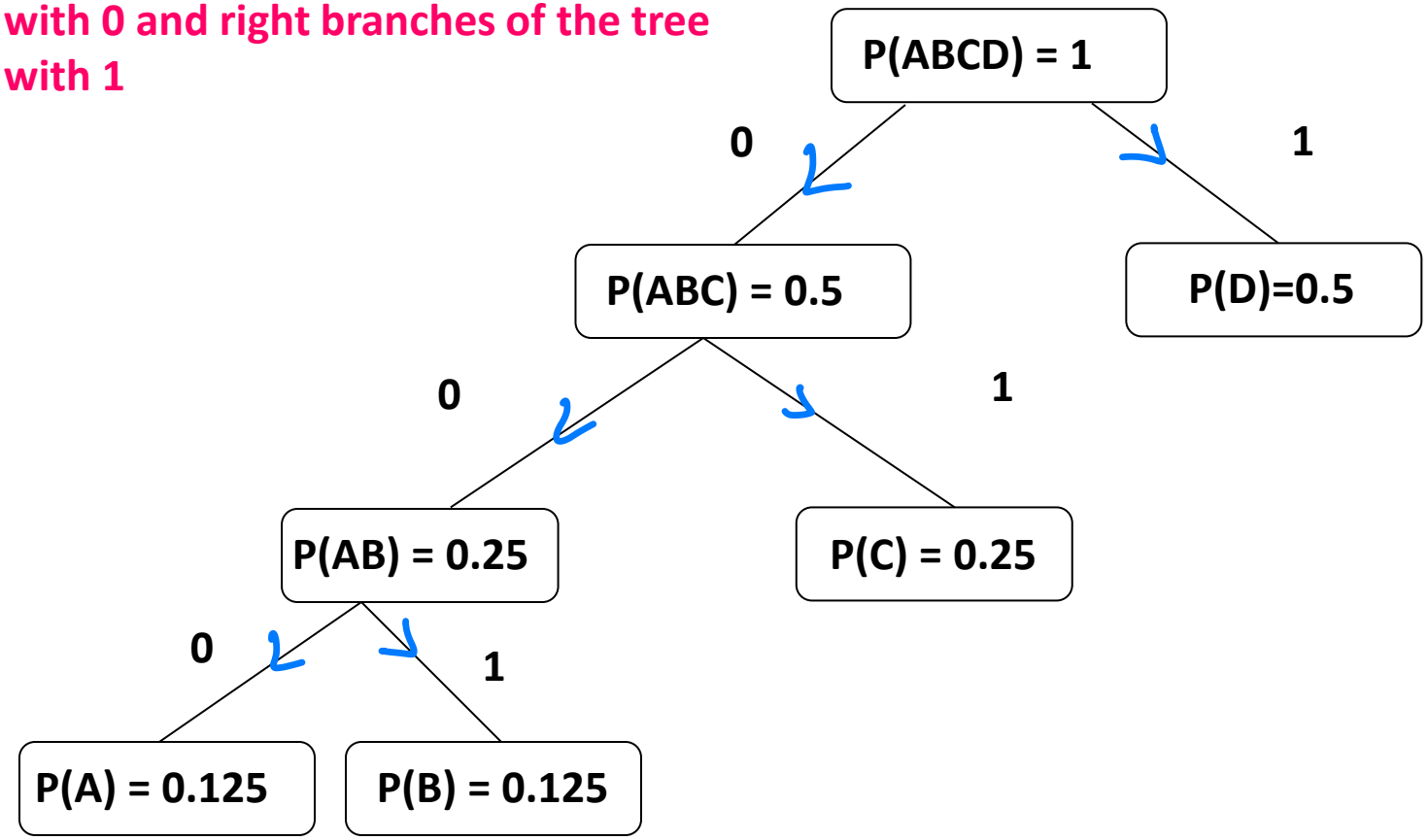
Step 2: Build a binary tree from left to right

Policy: always connect two smaller nodes together (e.g., P(A) and P(B) had both probabilities that were smaller than P(C), Hence those two connect first.



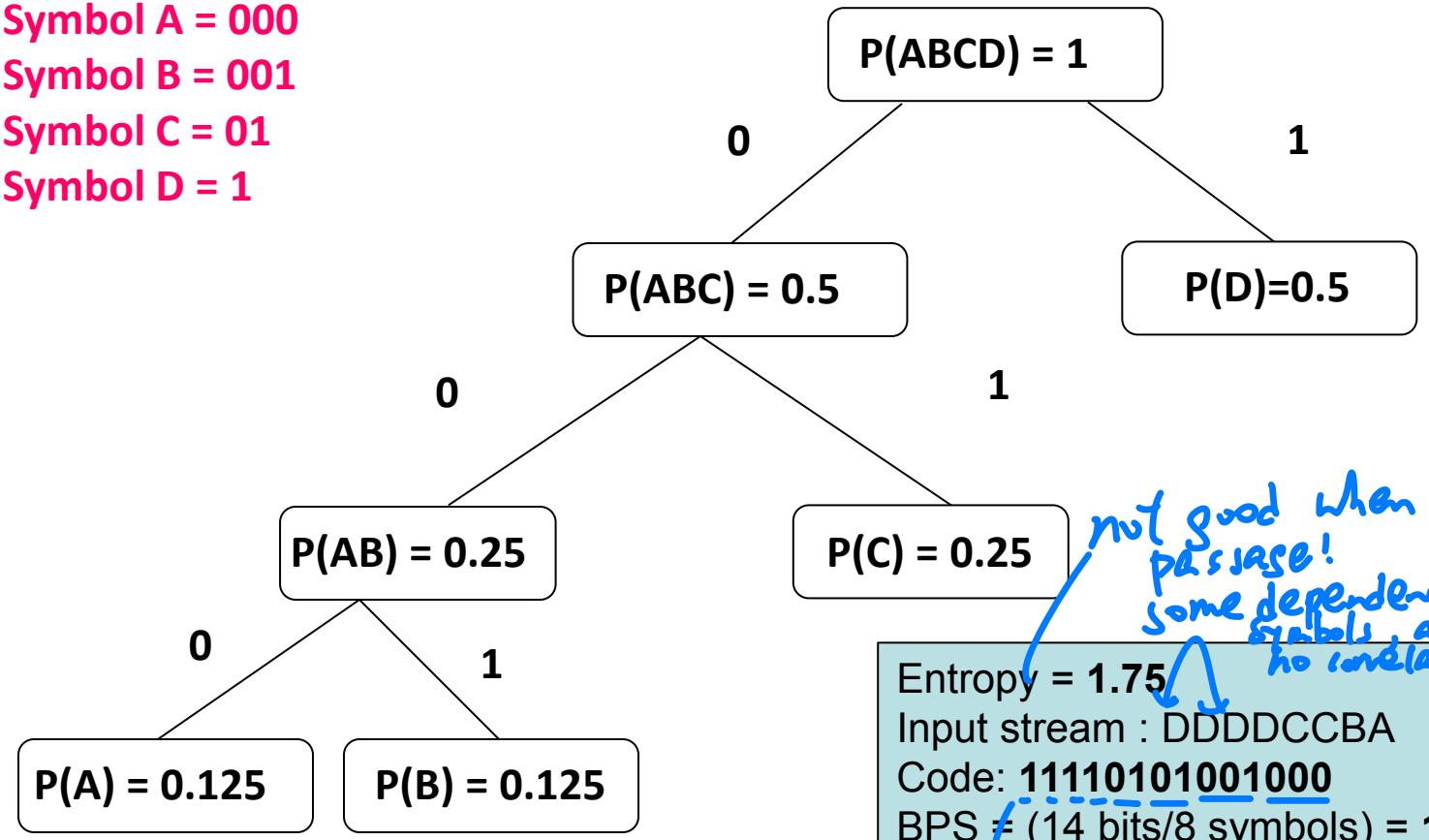
Huffman Encoding (Example)

Step 3: label left branches of the tree with 0 and right branches of the tree with 1



Huffman Encoding (Example)

Step 4: Create Huffman Code
Symbol A = 000
Symbol B = 001
Symbol C = 01
Symbol D = 1



Entropy = **1.75**
Input stream : DDDDCCBA
Code: **111101010001000**
BPS = (14 bits/8 symbols) = **1.75**

not good when encode passage! some dependency of symbols, assume no correlation!

Immediately decodable!

Huffman Coding

- Construct decoding tree based on encoding table.

- Read coded message bit-by-bit:
  - Traverse the tree top to bottom accordingly.
  - When a leaf is reached, a codeword was found corresponding symbol is decoded

"Global" English frequencies table:

Letter	Prob.	Letter	Prob.
A	0.0721	N	0.0638
B	0.0240	O	0.0681
C	0.0390	P	0.0290
D	0.0372	Q	0.0023
E	0.1224	R	0.0638
F	0.0272	S	0.0728
G	0.0178	T	0.0908
H	0.0449	U	0.0235
I	0.0779	V	0.0094
J	0.0013	W	0.0130
K	0.0054	X	0.0077
L	0.0426	Y	0.0126
M	0.0282	Z	0.0026
Total: 1.0000			

Problems with Huffman Coding

- Recall the rate of a code $R = \sum_{i=1}^{n} p(x_i) l(x_i)$ and the entropy of a source $H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 (p(x_i))$
- Entropy is achieved by a code only if $l(x_i) = \log_2 \frac{1}{p(x_i)}$
- But since Huffman coding can only assign an integer number of bits to each symbol, we can only hope for optimality when $p(x_i)$ is a power of 2: $p(x_i) = 2^{-l_i}$
- Huffman coding requires knowledge of the symbol probabilities. In practice, this means an algorithm with two passes.
- Also, it ignores any dependency among symbols (only good for memoryless sources).
- Huffman coding achieves a rate R satisfying:

$$H(X) \le R < H(X) + 1$$

which can be pretty bad.

Extended Huffman Coding

- Extended Huffman coding solves many of the problems of Huffman coding.
- It attempts to model the dependency of consecutive symbols.
- The idea is very simple: take groups of *n* symbols and treat that supersymbol just as a symbol and employ Huffman coding as usual.
- This means that the rate achieved will satisfy:

$$nH(X) \le R(extended) < nH(X)+1$$

or, equivalently,

$$H(X) \le R < H(X)+1/n$$

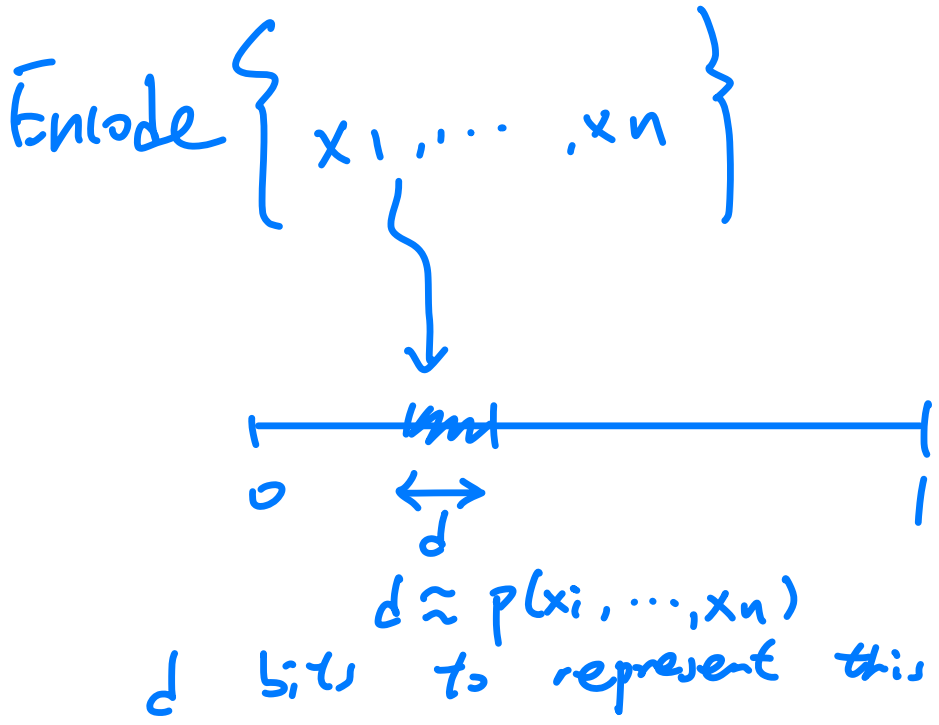
which is pretty good!

- However, the number of symbols grows exponentially with *n*!
- In addition, the performance depends on the previous estimation on the probabilities, which in practice can be bad.

Arithmetic Coding

- Huffman coding requires grouping of symbols to work well, however, this approach becomes impractical for long sequences of symbols.
- Arithmetic coding solves this problems. It is able to assign a codeword to a particular long sequence without having to generate codes for all sequences of that length.
- Represents the sequence $\{x_1,...,x_n\}$ by a subinterval of interval [0,1).
- Width of the subinterval approximately equal to the prob. of the sequence $f_x(\{x_1,...,x_n\})$.
- Subinterval can be determined by recursive subdivision algorithm.
- Represent sequence by shortest binary fraction in the subinterval.
- Subinterval of width $f_x(\{x_1,...,x_n\})$ is guaranteed to contain one number that can be represented by L binary digits with

$$L \approx -\log_2 f_x(\{x_1,...,x_n\})$$



$d \approx p(x_1, \cdots, x_n)$

d bits to represent this number.

We want to use $\sim \log_2 \frac{1}{p(x_1, \cdots x_n)} = \log_2 \frac{1}{d}$

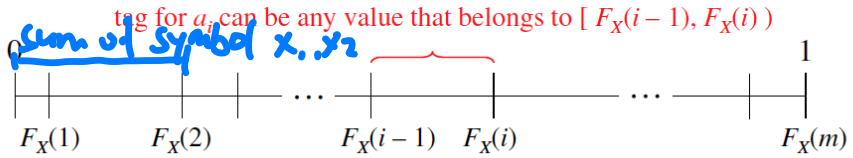
- pick a number in this interval

- Given a source alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$, a random variable $X(a_i) = i$, and a probability model $\mathcal{P}$: $P(X = i) = P(a_i)$. The CDF is defined as:

$$F_X(i) = \sum_{k=1}^{i} P(X = k).$$

Definition of CDF = $P(x \le k)$

- CDF divides $[0, 1)$ into disjoint subintervals:

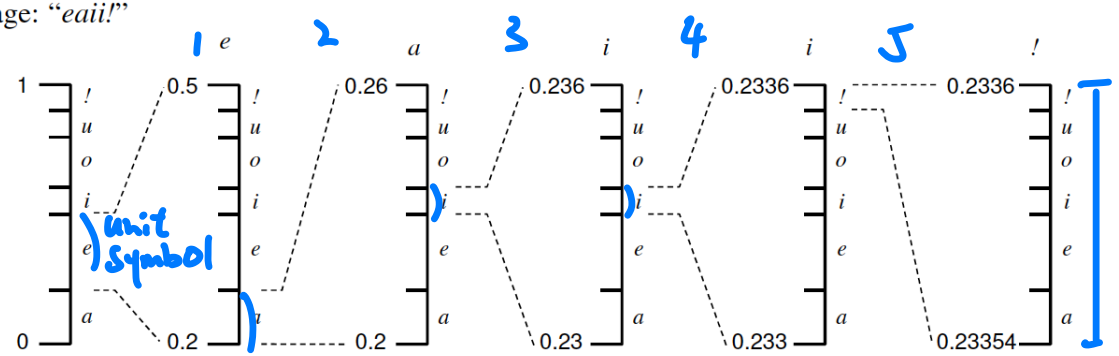


- In arithmetic coding, each symbol is mapped to an interval

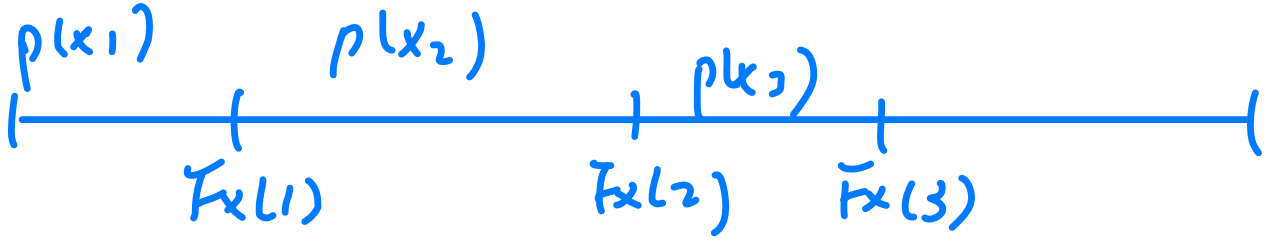
Symbol	Probability	Interval
a	.2	[0, 0.2)
e	.3	[0.2, 0.5)
i	.1	[0.5, 0.6)
o	.2	[0.6, 0.8)
u	.1	[0.8, 0.9)
!	.1	[0.9, 1.0)

?

message: "*eaii!*"



Recursive, 不用考慮 how many symbols, low complexity


p(x1)
p(x2)
p(x3)
Fx(1)
Fx(2)
Fx(3)

Arithmetic Encoding

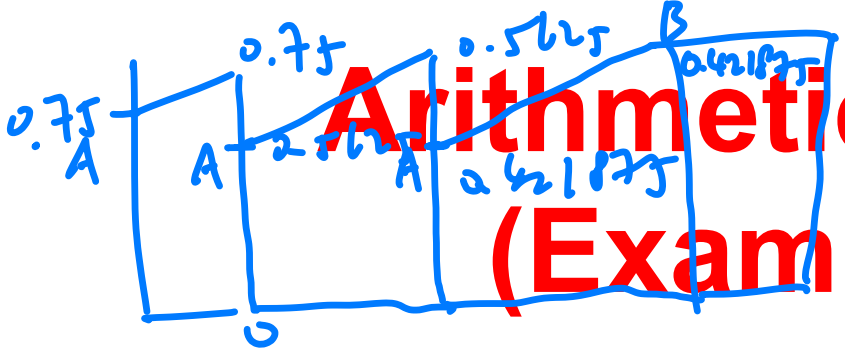
- Arithmetic encoder maintains two numbers, *low* and *high*, which represent a subinterval [*low*, *high*) of the interval [0, 1).
- Initially, *low* = 0 and *high* = 1.
- Symbols of the input sequence are numbered from 1 to *n* and symbol *i* has probability *P*(*i*).
- Scale (Narrow-down) the remaining intervals for the *s*-th symbol:
  - low_bound = $\sum_{i=1}^{s-1} P(i)$
  - high_bound = $\sum_{i=1}^{s} P(i)$
  - range = *high* – *low*
  - *low* = *low* + range * low_bound
  - *high* = *low* + range * high_bound

Arithmetic Decoding

- Arithmetic decoder also maintains two numbers, *low* and *high*, and knows the initial interval [0, 1).
- Initially, *low* = 0 and *high* = 1.
- Symbols are numbered from 1 to *n* and symbol *i* has probability $P(i)$.
- *value* is the arithmetic code to be decoded:
  - Find *s* such that

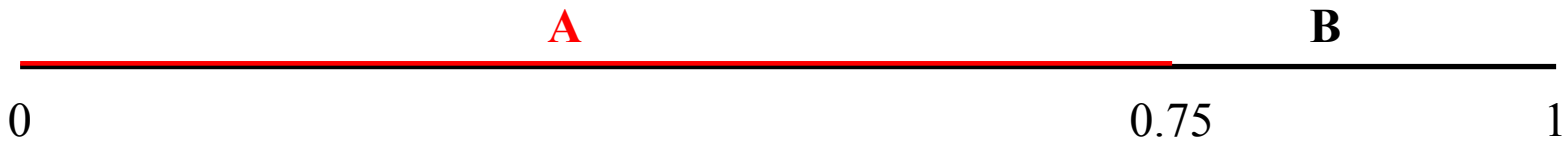
$$\sum_{i=1}^{s-1} P(i) \le \frac{value - low}{high - low} \le \sum_{i=1}^{s} P(i)$$

  - Return symbol *s*
  - Perform the same range-narrowing step of the encoding step

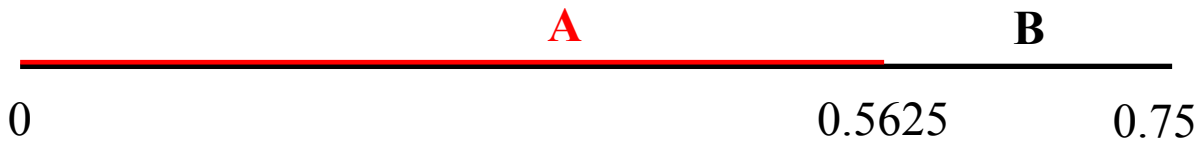


Arithmetic Coding (Example A)

- Start with a simple binary example. Supposing a sequence **AAAB** and encode/decode it with arithmetic coding.
- P(A) = 0.75 P(B) = 0.25. Initialize with *low* = 0 *high* = 1.
- First entry A → *low* = 0 *high* = 0.75

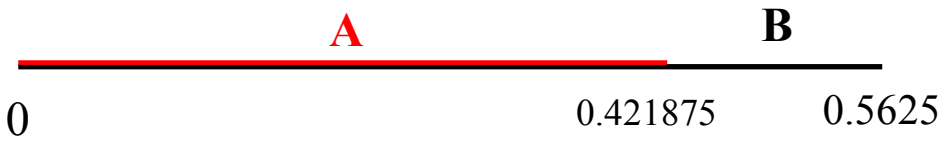


- Second entry A → *low* = 0 *high* = 0.5625 (0.5625=0.75*0.75)



Arithmetic Coding (Example A)

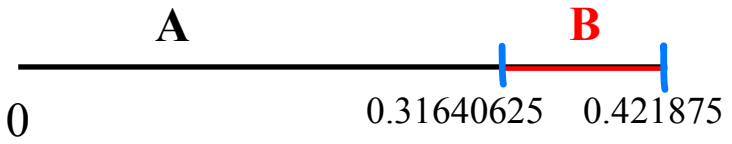
- Start with a simple binary example. Supposing a sequence **AAAB** and encode/decode it with arithmetic coding.
- Third entry A → *low* = 0 *high* = 0.421875 (0.421875=0.5625*0.75)



- Fourth entry B → *low* = 0.31640625 *high* = 0.421875 (0.31640625=0.421875-0.421875*0.25)

3/8

0.011 ✓



[0.31640625, 0.421875)

We select 0.375 in this interval

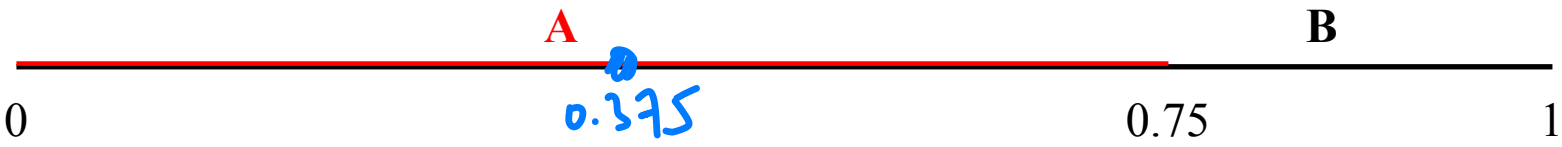
binarized as 0.011

encoded as 011 takes 3 bit ✓

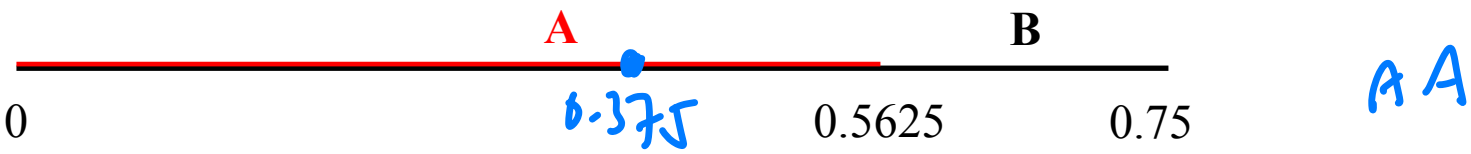
Decode:

Arithmetic Coding (Example A)

- Then we try to decode 0.011 → 0.375 with P(A) = 0.75 P(B) = 0.25



- 0.375 falls in [0, 0.75) → First entry A

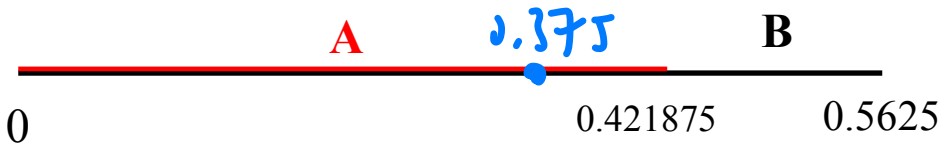


A A

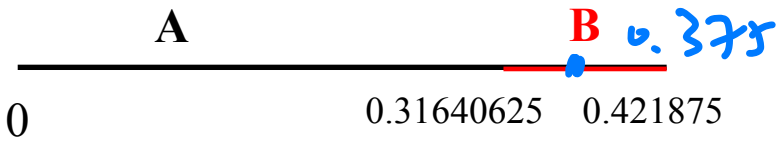
- 0.375 falls in [0, 0.5625) → Second entry B

Arithmetic Coding (Example A)

- Then we try to decode 0.011 → 0.375 with P(A) = 0.75 P(B) = 0.25



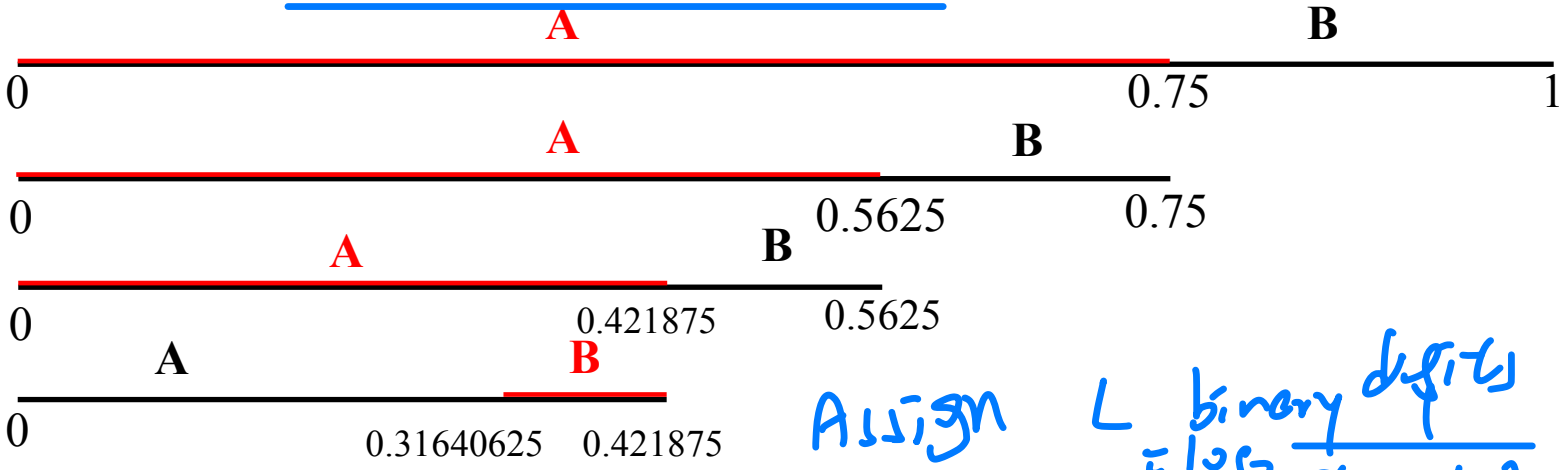
- 0.375 falls in [0, 0.421875) → Third entry A



- 0.375 falls in [0.31640625, 0.421875) → Fourth entry B
- Decoded result: AAAB

Arithmetic Coding (Example A)

- **Thinking 1**: Why arithmetic coding works?



- Since it assigns wider interval for higher-frequency letter → wider interval means we can find a shorter binary decimal → shorter bit length
- **Thinking 2**: How to find the optimal binary decimal from the interval?
- Find the same part in the both ends of the interval, then add an '1'.
- 0.31640625: 0.01010001 → 0.011
- 0.421875: 0.011011

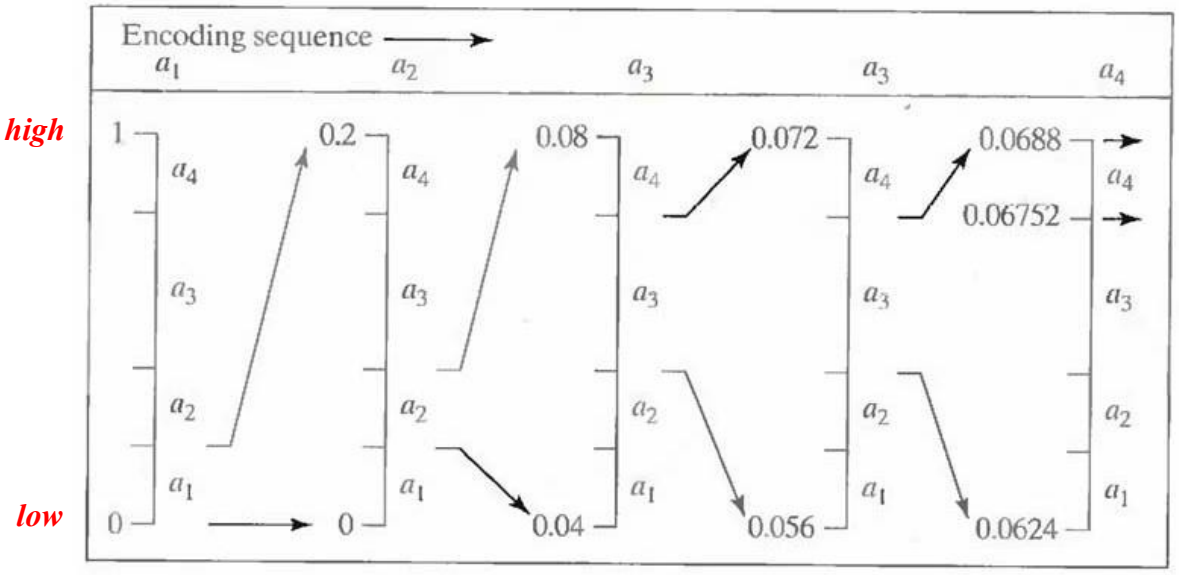
Arithmetic Coding (Example B)

- Given a four-letter alphabet $\{a_1, a_2, a_3, a_4\}$ with $P(a_1) = 0.2, P(a_2) = 0.2, P(a_3) = 0.4, P(a_4) = 0.2$. Consider encoding the string $a_1 a_2 a_3 a_3 a_4$.
- Initialization: *low* = 0 and *high* = 1.
- The interval [0, 1) is initially subdivided into four regions based on the probabilities of each symbol.

Source Symbol	Probability	Initial Subinterval
a_1	0.2	[0.0, 0.2)
a_2	0.2	[0.2, 0.4)
a_3	0.4	[0.4, 0.8)
a_4	0.2	[0.8, 1.0)

Arithmetic Coding (Example B)

- As a_1 is the first symbol, the encoder narrows the interval to [0, 0.2). Then, [0, 0.2) is expanded to the full height and its end points are labeled by the values of the narrowed range, which is divided with the probabilities. The process repeats until the end of string $a_1a_2a_3a_3a_4$.



Final interval: [0.06752, 0.0688)

We can send 0.068

decimal digit!

Arithmetic Coding (Example B)

- With Arithmetic coding, we can use three decimal digits, i.e., 0.068, to represent the five-symbol message $a_1 a_2 a_3 a_3 a_4$.

- This translates into 0.6 decimal digits per source symbol and compares favorably with the entropy of the source, which is 0.58 decimal digits.

- As the length of the sequence being encoded increases, the resulting arithmetic code approaches the data compression limit.

Arithmetic Coding (Example B)

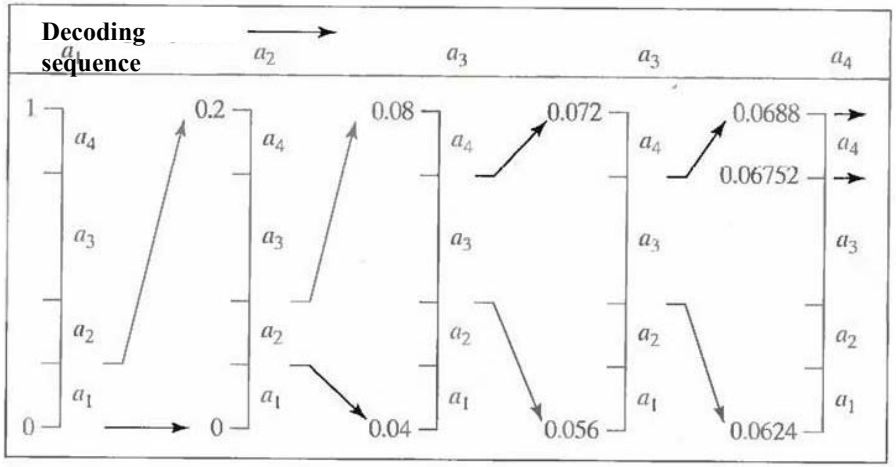
- Suppose we have to decode 0.068.
- The decoder needs symbol probabilities, as it simulates what the encoder must have been doing.
- The decoder starts with *low* = 0 and *high* = 1 and divides the interval exactly in the same manner as the encoder.

Source Symbol	Probability	Initial Subinterval
a_1	0.2	[0.0, 0.2)
a_2	0.2	[0.2, 0.4)
a_3	0.4	[0.4, 0.8)
a_4	0.2	[0.8, 1.0)

Arithmetic Coding (Example B)

- 0.068 falls in [0, 0.2)
- 0.068 falls in [0.04, 0.08)
- 0.068 falls in [0.056, 0.072)
- 0.068 falls in [0.0624, 0.0688)
- 0.068 falls in [0.06752, 0.0688)

a_1
a_2
a_3
a_3
a_4



Output: $a_1 a_2 a_3 a_3 a_4$

Arithmetic vs. Huffman

nowadays very popular

- Arithmetic coding requires less memory, as symbol representation is calculated on the fly.
- Arithmetic coding is more suitable for high performance models, where there are confident predictions.
- Huffman decoding is generally faster than Arithmetic decoding.
- In Arithmetic coding it is not easy to start decoding in the middle of the stream, while in Huffman coding we can use "starting points".
- In large collections of text and images, Huffman coding is likely to be used for the text, and Arithmetic coding for the images.

LZW Coding

- What if the probability for each alphabet is unknown? Lempel-Ziv-Welch coding solves this general case, where only a stream of bits is given. No need for two passes.
- In simple Huffman coding, the dependency between the symbols is ignored, while in the LZW, these dependencies are identified and exploited to perform better encoding.
- When all the data is known (alphabet, probabilities, no dependencies), it's best to use Huffman (LZW will try to find dependencies which are not there.)

LZW coding

use this to represent data!

- LZW creates its **own dictionary** (strings of bits), and replaces future occurrences of these strings by a shorter position string.

- Parses source input (in binary) into the shortest distinct strings:

input! 1011010100010 ->1, 0, 11, 01, 010, 00, 10

some data!

- Each string includes a prefix and an extra bit (010 = 01 + 0), therefore encoded as: (prefix string place + extra bit)

- For example, 010 is encoded as the “location index of 01”+0.

LZW Algorithm

1. Initialize the dictionary to contain an empty string (D={ }).
2. Find "W", the longest block in input string which appears in D.
3. Find "B", first symbol in input string after W
4. Encode W by its index in the dictionary, followed by B
5. Add W+B to the dictionary.
6. Go to Step 2.

Why this can achieve?

Input string: 1 0 1 1 0 1 0 1 0 0 0 1 0

W	ø	ø	1	0	01	0	1
B	1	0	1	1	0	0	0

what is index in D such that entry is 01? is 2!

Dictionary D	
Index	Entry
0	ø
1	1
2	0
3	11
4	01
5	010
6	00
7	10

← longest! Encoded string:

Pairs:	(0,1)	(0,0)	(1,1)	(2,1)	(4,0)	(2,0)	(1,0)
Encoding:	0001	0000	0011	0101	1000	0100	0010

成3就是4!

00010000001101011000010000010

Huffman vs. Lempel-Ziv

Compression comparison

Compressed to (percentage):	Lempel-Ziv (unix gzip)	Huffman (unix pack)
html (25k) *Token based ascii file*	**20%**	65%
pdf (690k) *Binary file*	**75%**	95%
ABCD (1.5k) *Random ascii file*	33%	**28.2%**
ABCD(500k) *Random ascii file*	29%	**28.1%**

$ABCD - \{p_A = 0.5, p_B = 0.25, p_C = 0.125, p_D = 0.125\}$

Lempel-Ziv is asymptotically optimal

especially for long substring!