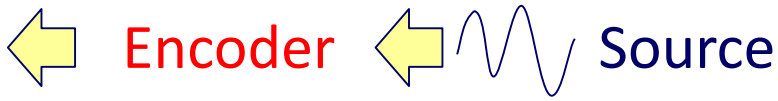
# *Compact Representation of Information*

**Objectives**: Learn how to encode symbols generated by information sources.




**Purpose of source encoding:**

- to represent the source in a way that is suitable for transmission (Recall the process of sampling and quantization)
- to remove source redundancy (make it more compact)

**Desirable Properties of a Source Encoder (in general)**

- as precise as possible (lossless or with a small controlled loss)
- as compact as possible (source compression)
- uniquely decodable.
- immediately (instantaneously) decodable

# Topics to be Covered in this Module

**Source Encoding:** Map sequences of source symbols (messages) into binary sequences with unique decodability.

**Topics to be covered**

- Construction of source codes (*binary* case)
- Prefix-free codes for discrete sources
- Theoretical limit of the "compression"
- Source coding theorem
- Related topics:
  - Fixed length encoding
  - variable length encoding
  - Kraft inequality,

# Words and Terms

First, consider the source encoding on a symbol-by-symbol basis only.

- *M: T*he set of symbols generated by an information source.
- For each symbol in *M*, associate a sequence over the code alphabet {0, 1}.
  - codewords : binary sequences associated to symbols in *M*
  - code: the set of codewords
  - Code alphabet: {0, 1} , i.e., binary code (in this course)
  - Source alphabet M={sunny, cloudy, rainy}, |M|=3.

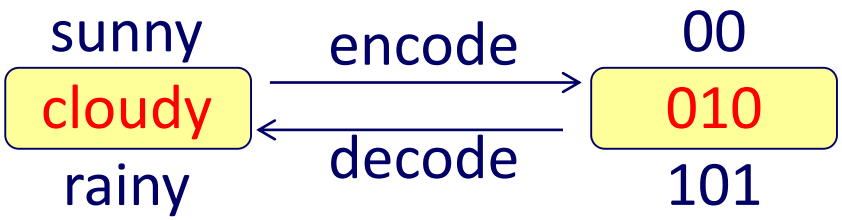
| *M* (Source) | Codewords) |
|---|---|
| sunny | 00 |
| cloudy | 010 |
| rainy | 101 |

code *C* = {00, 010 and 101}

Remark: This is NOT an efficient code. Why?

# Encoding and Decoding

- encode ... to map a codeword for each source symbol
- decode ... to retrieve the source symbol given the codeword




- NO separation symbols between codewords;
  - 010 00 101 101 ... Not acceptable, 01000101101 ... OK

*Why?*

- {0, 1 and “space”} ... the alphabet have three symbols, not two

# Fixed Length Encoding

- Here a fixed number of digits (**r**) is assigned to every symbol in the source alphabet without regard to the probability of occurrence of the symbols.
- Choose **r** to be an integer that satisfies:

$$\log_2 M \leq r < \log_2 M + 1$$; M is the size of the alphabet

- Example: Let the possible source symbols be the set:

  S={red, blue, green, yellow, purple, magenta}; S= {R, B, G, Y, P,M}

- Here, M= 6 and **r= 3** bits/symbol. **$2.584 \leq r < 3.584$;**
- One possible encoding scheme is:

green → 010 Red → 000 yellow → 011 purple → 100

magenta → 101 blue → 001

**Question**: Can we do better than 3 bits/symbol for the fixed length code:

**Answer**: **Use the concept of extended source**

# Fixed Length Encoding

- **Example**: If in the previous example, 3 symbols are combined together to form a new (message). The new set of possible messages is:

**S'**={RRR, RRB, RMM,…,MRM}, **M'**= $M^3$ = 216; (New source alphabet)

- With **M'**=216, **r'=8** bits/three source symbols (i.e., b bits/message)
- r' satisfies the relation:

  **$\log_2 M' \le r' < \log_2 M' + 1$**; **$7.75 \le r' < 8.75$; bits/message**

  **$3\log_2 M \le r' < 3\log_2 M + 1$; Dividing both sides by 3, we get**

  **$\log_2 M \le r < \log_2 M + 1/3$;** **r=8/3=2.66 bits/source symbol**

  In general, if we combine n source symbols together, r satisfies

  **$\log_2 M \le r < \log_2 M + 1/n$**

- This says that as **n→ infinity**, **r → $\log_2 M$** bits/source symbol
- **This is the best that can be done with fixed length encoding**

# Fixed Length Encoding

## $\log_2 M' \le r < \log_2 M' + 1/n;$

- Let M=6; r: number of bits/symbol

| n<br>1 | M'<br>M'=6 | $\log_2 M'$<br>$\log_2 M'=2.58$ | $\log_2 M' \le r' < \log_2 M' + 1$<br>r'=3 bits/mess | r=r'/n<br>r=3 bits/sym |
|---|---|---|---|---|
| 3 | $6^3=216$ | log216=7.75 | r'=8 bits/mess | r=2.666 bits/sym |
| 4 | $6^4=1296$ | log1296=10.33 | r'=11 bits/mess | r=2.75 bits/sym |
| 6 | $6^6=46656$ | $\log M^6=15.48$ | r'=16 bits/mess | r=2.66 bits/sym |
| 8 | $6^8=6^8$ | $\log 6^8=20.64$ | r'=21 bits/mess | r=2.62 bits/sym |
| 10 | $6^{10}=6^{10}$ | $\log 6^{10}=25.8$ | r'=26 bits/mess | r=2.6 bits/sym |

As message size n becomes larger, r approaches $2.58= \log_2 6$
This is the best that can be done with fixed length encoding.

Question: Can we do better than $\log_2 M$

Answer: Yes, using variable length encoding

# Variable Length Source Encoding

Lecture 10

- **Basic Idea**: Symbols with high probability of occurrence should have shorter codewords than symbols with low probability of occurrence in order to reduce the average number of bits/symbol.
- A variable length source code C assigns to each source symbol **s** a codeword **C(s)** of length **l(s)**.
- **Example: Let the source alphabet be the set S ={ a, b, c}**

  C(a) = 0; l(a) = 1 bit

  C(b) = 10; l(b) = 2 bits

  C(c) = 11; l(c) = 2 bits.

# Uniquely Decodable Codes

- A code must be uniquely decodable, which means
  - any two distinct symbols should have distinct code-words.
  - Different symbol sequences should be encoded to different binary sequences.

| | $C_1$ | $C_2$ | $C_3$ | $C_4$ |
|---|---|---|---|---|
| $a_1$ | 00 | 0 | 0 | 0 |
| $a_2$ | 10 | 01 | 10 | 10 |
| $a_3$ | 01 | 011 | 11 | 11 |
| $a_4$ | 11 | 111 | 01 | 0 |
| | yes | Yes | no | no |

with the code $C_3$...

$a_1\ a_3\ a_1$ → 0110 ← $a_4\ a_2$

Same binary sequence

Different sequences

with code $C_2$

$a_1\ a_3\ a_1$ → 00110

$a_4\ a_2$ → 11101

Different Sequences

Different Sequences

# Uniquely Decodable Codes

| | $C_1$ | $C_2$ |
|---|---|---|
| $a_1$ | 00 | 0 |
| $a_2$ | 10 | 01 |
| $a_3$ | 01 | 011 |
| $a_4$ | 11 | 111 |

A Problem with $C_2$ : consider a scenario of using $C_2$

- Sequence $\{a_1, a_4, a_4, a_1\}$ is encoded to 01111110.
- The receiver may decode it as:
  - (0)(111)(111)(0) decoded into $\{a_1, a_4, a_4, a_1\}$
  - *the code is **uniquely decodable**, but is **not instantaneously decodable***)
  - **What is the difference between the two?**
  - **Uniquely decodable** means we can retrieve the source symbols from the encoded binary bits, but the first symbol cannot be decoded without reading all the binary sequence all the way to the end.
  - **Instantaneously decodable** means that once the code-word corresponding to a symbol is received, the symbol will be decoded instantaneously, without waiting for the next symbol

# Prefix-free codes for discrete sources

- **Prefix-free codes**: A simple class of uniquely decodable codes. They have the following advantages over other uniquely-decodable codes:

- The decoder can decode each codeword of a prefix-free code **immediately** on the arrival of the last bit in that codeword without waiting for the entire sequence of bits to arrive.
- Given a probability distribution on the source symbols, it is easy to construct a prefix-free code of **minimum expected length**.
- If a uniquely-decodable code **exists** with a certain set of codeword lengths, then a prefix-free code can be easily **constructed** with the same set of **lengths**.

# Prefix-free codes for discrete sources

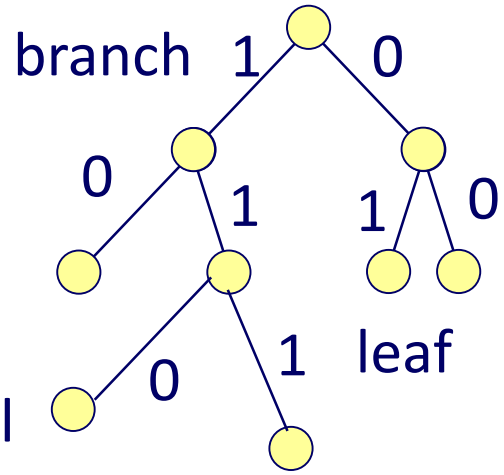
- **Definition**: A code is prefix-free if no codeword is a prefix of any other codeword.
- **Example**: The code $C_1$ is not prefix free. Why?
  - "0" is a prefix of "01" and "011"
  - "01" is a prefix of "011"
- **Example**: The code $C_2$ is prefix free.
- **Remark 1**: Every fixed-length code with distinct codewords is prefix-free.
- **Remark 2**: every prefix-free code is uniquely decodable

| | $C_1$ |
|---|---|
| $a_1$ | 0 |
| $a_2$ | 01 |
| $a_3$ | 011 |
| $a_4$ | 111 |

| | $C_2$ |
|---|---|
| $a_1$ | 0 |
| $a_2$ | 10 |
| $a_3$ | 11 |

# construction of prefix-free codes (*binary* case)

how to construct a prefix-free code with *M* codewords

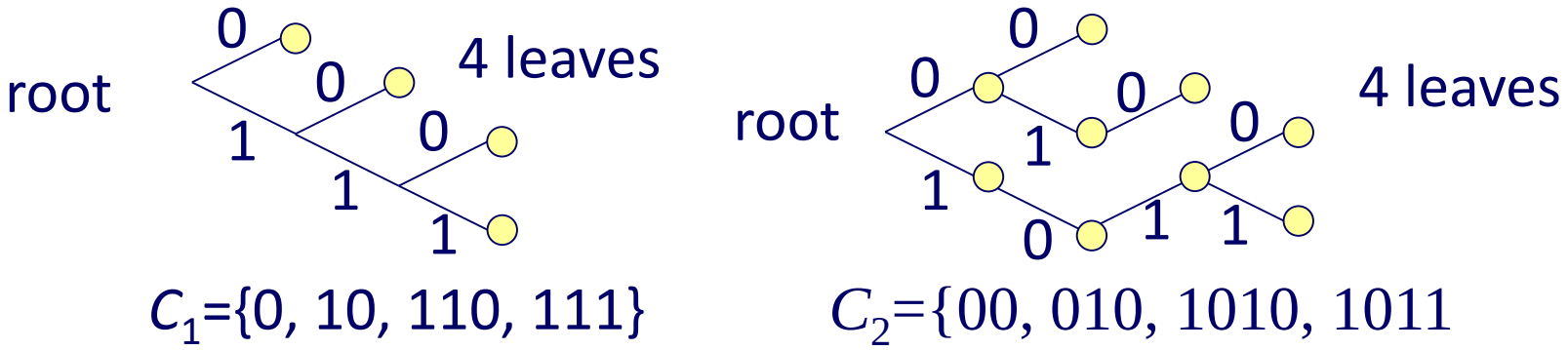



1. construct a *binary* tree *T* with *M* leaf nodes

1. for each branch of *T*, assign a label in {0,1 }
   - sibling branches cannot have the same label
3. for each of leaf nodes of *T*, traverse *T* from the root to the leaf, with concatenating labels on branches. The obtained sequence is the prefix-free codeword of the node.

➢ Here, we have 5 leaves. The codewords are: {00, 01,10, 111, 110}

# The "best" among Immediately Decodable Codes




$C_1$={0, 10, 110, 111}  $C_2$={00, 010, 1010, 1011

- *Lengths for* $C_1$={1, 2, 3, 3}; *Lengths for* $C_2$={2, 3, 4, 4};
- $C_1$ seems to give more compact representation than $C_3$.
- **Q**: Is there a more compact code than $C_1$ and $C_2$?
- **Q**: Can we construct prefix-free codes with the following lengths?
  - codeword length = [1, 1, 2, 2]?
  - codeword length = [1, 2, 2, 2]?
  - codeword length = [1, 2, 2, 3]?

**Q**: What is the criteria?

# The Kraft Inequality

- **Kraft inequality** is a test on the existence of prefix-free code with a given set of codeword lengths {l(s), s ε S}.
- **Theorem**: Every prefix-free code for an alphabet S with codeword lengths {l(s), s ε S} satisfies:

$$\sum_{s \in S} 2^{-l(s)} \leq 1$$

- **Conversely**, if the above inequality above holds, then a prefix-free code with lengths {l(s)} exists.
- The proof is omitted.

# The Kraft Inequality: Examples

**Example**: Can we construct a prefix-free code for the source with symbols {a, b, c} with lengths {1, 1, 2}

**Solution**: The Kraft inequality provides a test on the existence of prefix-free codes with a given set of codeword lengths.

$$2^{-1} + 2^{-1} + 2^{-2} = 1.25 > 1$$

The answer is **NO**. Recall the code {0, 1, 01} or code {0, 1, 10}

- **Example**: Can we construct a prefix-free code for the source with symbols {a, b, c} and lengths {1, 2, 2}
- **Solution**: We test the existence of the code using the Kraft inequality .

$$2^{-1} + 2^{-2} + 2^{-2} = 1$$

The answer is **YES**. (recall the code 0, 10, 11)

# Prefix-free codes for DMS

- Let l(x) be the length of the codeword for letter x ∈ X . The probability of symbol x is P(x), for all x
- Then, L(X) is a random variable. The mean value of L(X) is

$$L = \sum_{x} l(x)P(x)$$

- L is the average number of bits /source symbol.
- **Example:** A source emits one of four possible symbols {a, b, c, d} every unit of time with probabilities:

  P(a) = 0.4  P(b)=0.3

  P(c) = 0,2  P(d)= 0.1

- Three different prefix-free codes are proposed. We compute the average number of bits/symbol for each code.

# Example: computing the average codeword length

| symbol | probability | $C_1$ | $C_2$ | $C_3$ |
|---|---|---|---|---|
| *a* | 0.4 | 0 | 111 | 00 |
| *b* | 0.3 | 10 | 110 | 01 |
| *c* | 0.2 | 110 | 10 | 10 |
| *d* | 0.1 | 111 | 0 | 11 |

All three codes are prefix-free.

**The average (expected) value of the length of each code is:**

- **$C_1$: $L_1$ = 0.4 × 1+ 0.3 × 2+ 0.2 × 3+ 0.1 × 3 = 1.9 bits/symbol**
- **$C_2$: $L_2$ = 0.4 × 3+ 0.3 × 3+ 0.2 × 2+ 0.1 × 1 = 2.6 bits/symbol**
- **$C_3$:$L_3$ =: 0.4 × 2+ 0.3 × 2+ 0.2 × 2+ 0.1 × 2 = 2.0 bits/symbol**

$C_1$ gives the most compact representation in typical cases.

**Question**: Can we get an average length smaller than that of $C_1$?

# Prefix-free codes with minimum average length

- Let X ={1, 2,…, M} be the source alphabet with a known probability mass function (pmf) P(X=x)= {$p_1$, $p_2$, … $m_M$}
- Let l(x) be the length of a codeword in a prefix-free code for letter x ∈ X . These lengths are unknown and they should be integers.
- **Objective**: Choose the lengths l(x) so that the average length of codeword is minimized and at the same time maintain the prefix-free property of the code.
- The problem then becomes

$$\text{Given } p(1), p(2), \ldots, p(M), \text{find } l(1), l(2), \ldots, l(M) \text{ that}$$

$$\text{minimize } L = \sum_x p(x) l(x)$$

$$\text{subject to: } \sum_x 2^{-l(x)} \leq 1; \text{ Kraft Inequality}$$

# Prefix-free codes with minimum average length

- The problem:

$$\text{Find } l(1), l(2), \ldots, l(\mathrm{M}) \quad \text{that}$$

$$\text{Minimize } \mathrm{L} = \sum_x \mathrm{p}(x) l(x) \quad \text{Subject to: } \sum_x 2^{-l(x)} \le 1; \ \text{K.I.}$$

- This is a constrained optimization problem which can be solved using Lagrange method. The derivation is carried out on the next slide.
- **The Optimum Solution** to the problem (lifting the condition that l(x) is an integer) is:

$$l(x) = -\log p(x)$$

$$L_{\min} = -\sum_x p(x) \log p(x) = H(X)$$

# Optimal Codes

Minimize $\bar{l} = \sum p_i l_i$ under the constraint $\sum 2^{-l_i} \leq 1$.

*Average codeword length [bits/codeword]*

*Kraft's Inequality*

Disregarding integer constraints, we get that
$J = \sum p_i \cdot l_i + \lambda \sum 2^{-l_i}$ should be minimized.

J: Objective Function
$\lambda$: Lagrange Multiplier
(to be determined)

Differentiate: $\frac{\delta J}{\delta l_i} = p_i - \lambda 2^{-l_i} \ln 2$

$\frac{\delta J}{\delta l_i} = 0 \Rightarrow 2^{-l_i} = \frac{p_i}{\lambda \ln 2}$

# Optimal Codes

Minimize $\bar{l} = \sum p_i l_i$ under the constraint $\sum 2^{-l_i} \leq 1$.

Differentiate: $\frac{\delta J}{\delta l_i} = p_i - \lambda 2^{-l_i} \ln 2$

$\frac{\delta J}{\delta l_i} = 0 \Rightarrow 2^{-l_i} = \frac{p_i}{\lambda \ln 2}$

Kraft's Inequality: $\sum 2^{-l_i} = 1$ $\Rightarrow \sum \frac{p_i}{\lambda \ln 2} = \frac{1}{\lambda \ln 2} = 1$

$$p(x)=2^{-l(x)} \qquad l(x) = -\log p(x)$$

$$\bar{L}_{min} = \sum_{x=1}^{M} l(x)p(x) = -\sum_{x=1}^{M} p(x)\log p(x) = H(X)$$

# Prefix-free codes with minimum average length

- $\bar{L}_{min} = \sum p(x)l(x) = -\sum p(x)logp(x) = H$
- The best average length per symbol is the source entropy and this is achieved when
- symbol x with prob. $p(x)$ is assigned a length $l(x) = -logp(x)$
- But, $l(x)$ is not necessarily an integer, in general.
- Then, we take the closest higher integer value, i.e., choose
  - $l(x) = -logp(x) + s(x)$, where $0 < s(x) < 1$,
- Hence, the practical bounds on the average length
  - $-\sum p(x)logp(x) < \bar{L} < \sum p(x)logp(x) + \sum p(x)s(x)$
  - $-\sum p(x)logp(x) < \bar{L} < \sum p(x)logp(x) + 1$
  - $\boldsymbol{H < \bar{L} < H + 1}$

# The Source Coding Theorem

- Assume that the source $X$ is memory-less.
- If we take massages of length n symbols and encode them, the bounds on the average length per message becomes:

$$nH(X) \le n\overline{L} \le nH(X) + 1$$

From which we get the bounds on the average length per symbol as:

$$H(X) \le \overline{L} \le H(X) + \frac{1}{n}$$

- *We can come arbitrarily close to the entropy as the sise of the message n increases!*

# Prefix-free codes with minimum average length

$$l(x) = -\log P(x) \qquad L_{\min} = -\sum_{x} P(x) \log P(x) = H(X)$$

- The lower bound on the average length per symbol is achieved when:

$$P(x) = 2^{-l(x)}$$

- **Example**: Let P(a)=0.5 , P(b)=0.25 , P(c)=0.25
- Note that: Let P(a)=0.5 = $2^{-1}$, P(b)=0.25 = $2^{-2}$ , P(c)=0.25 = $2^{-2}$ .
- l(a) = 1; l(b)=2; l(c)=2;
- The Code: {0, 10, 11} has an average length = Entropy H

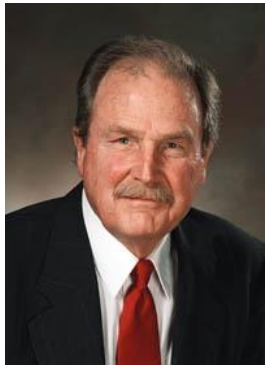
# Creating a Code: The Data Compression Problem

Lecture 11

- Assume a source with an alphabet X and known symbol probabilities $\{P(x)\}$.
- **Basic properties needed for source coding**
  - as precise as possible (lossless or with small loss)
  - as compact as possible
  - Prefix-free code (instantaneously decodable)
- **Goal:** Choose the codeword lengths as to minimize the average number of bits per symbol $\overline{L} = \sum l(x)P(x)$
- **Restriction**: We want an instantaneous code, so $\sum 2^{-l_x} \leq 1$ (Kraft inequality) must be valid
- **Solution** : at least in theory, we must have $l(x) = -logP(x)$

# Huffman Code

- Huffman coding is a technique used to compress files for transmission
- Works well for text and fax transmissions
- Huffman algorithm gives a clever way to construct a code with small average codeword length.
- Uses statistical coding
  - **symbols with high probability have shorter code words**
- The idea is to assign to each symbol in the source alphabet a number of binary digits equal roughly to the amount of information carried by that symbol $\boldsymbol{l(x) = -logP(x)}$
- If P(i) ≥ P(j), then $\boldsymbol{l(i) \leq l(j)}$.
- The end result is a code whose average length approaches the entropy limit; [ H(X) ≤ L ≤ H(X) +1 ].
- Huffman code satisfies this condition H(X) ≤ L ≤ H(X) +1 (for a message of length n=1).



*David Huffman*
*1925-1999*

# Huffman Encoding Algorithm

1. Arrange the source symbols in a decreasing order of probability. (**sorting stage**)
2. The last two symbols of lowest probability are assigned a 0 and 1. This step is referred to as a **splitting stage**.
3. The probability of the last two symbols are tied together to form a new symbol with prob. = sum of prob. of last two symbols (**merge stage**)
4. Arrange the new set of symbols in a decreasing order of prob. (**sorting stage**)
5. The last two symbols of lowest probability are assigned a 0 and 1 (**splitting stage**)

# Huffman Encoding Algorithm

6. The probability of the new last two symbols are tied together to form another new symbol with prob. = sum of prob. of last two symbols.
7. The procedure is repeated until we end up with only two symbols. Assign to them the digits 0 and 1.
8. The code for each source symbol is found by working backward and tracing the sequence of 0's and 1's assigned to that symbol and its successors.
9. **Result: Huffman code is an optimal code**

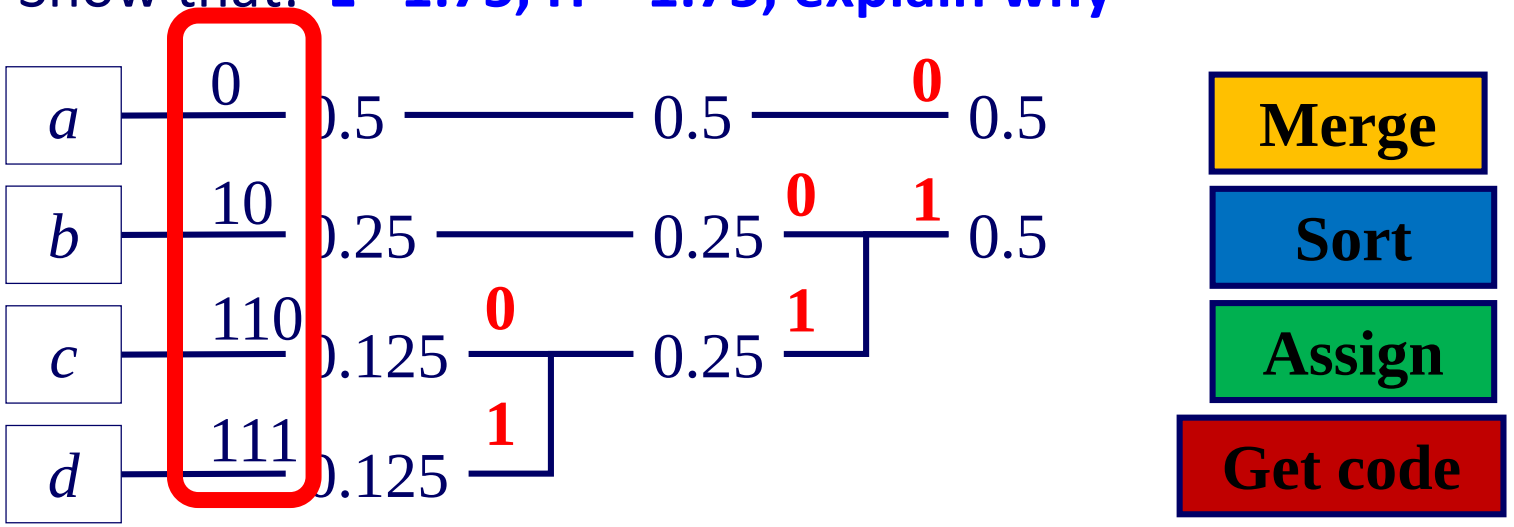
# Huffman Coding

- Two-step algorithm:

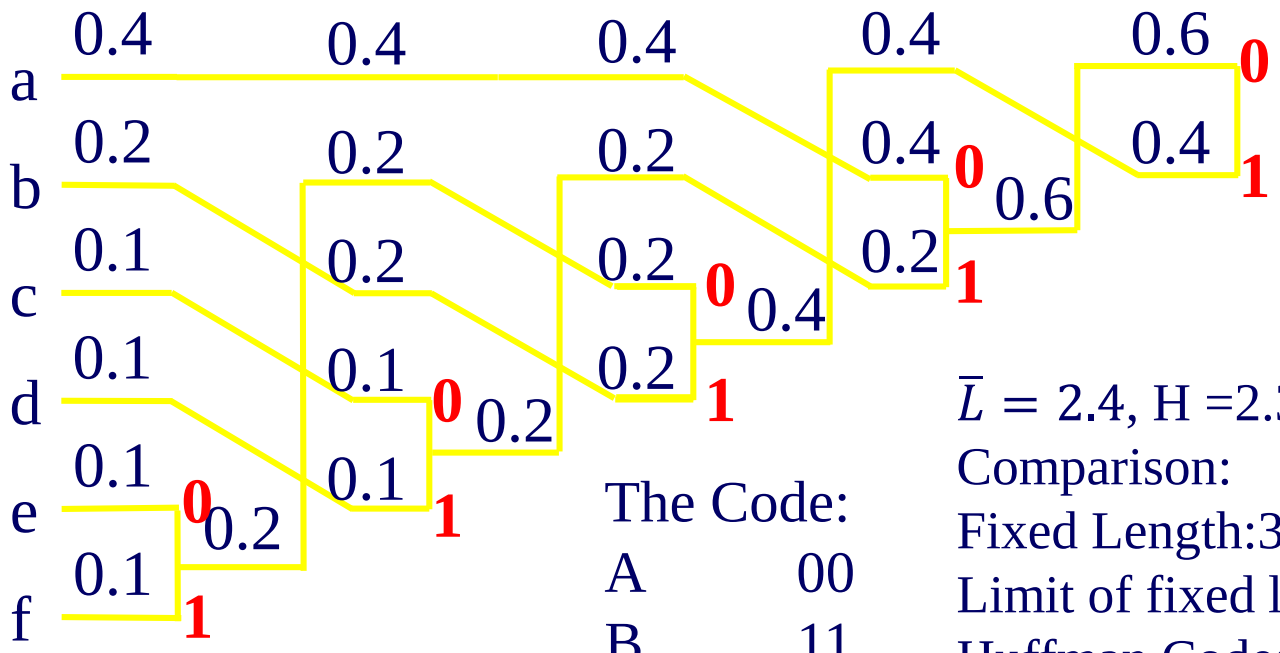
  Iterate:

  - Merge the least two probable symbols.
  - Arrange symbols in a decreasing order of probability (sort)

1. Assign binary bits to each source symbol (codewords)
2. **Example:** Find the Huffman code for a source with probabilities {0.5, 0.25, 0.125, 0.125}.
3. Show that: **L= 1.75, H = 1.75; explain why**



# Huffman Code Example: P(X)= {0.4, 0.2, 0.1, 0.1, 0.1, 0.1}




The Code:

| | |
|---|---|
| A | 00 |
| B | 11 |
| C | 010 |
| D | 011 |
| E | 100 |
| F | 101 |

$\bar{L} = 2.4$, H =2.32 (bits/symbol)

Comparison:

Fixed Length:3 bits/symbol

Limit of fixed length:$\log_2 6$=2.58

Huffman Code: 2.4

Limit of variable length: 2.32

# Coding for Extended Information Sources

The Huffman code is the best symbol-by-symbol code, but...

- the average code length $\overline{L} \geq 1$
- not good for encoding binary information sources (on a bit by bit basis). The average is 1 bit regardless of the bit probabilities

| symbol | prob. | $C_1$ | $C_2$ |
|---|---|---|---|
| A | 0.8 | **0** | **1** |
| B | 0.2 | **1** | **0** |
| average | | 1.0 | 1.0 |

If we encode several symbols in a block, then...

- $\overline{L}$ (per symbol) can be improved as we shall see next

# Example: Coding for Binary DMS Information Sources

| | prob. | codeword |
|---|---|---|
| A | 0.8 | 0 |
| B | 0.2 | 1 |

- The Huffman Code {0, 1}
- $\bar{L}$=0.8 × 1+ 0.2 × 1 = 1.0 bit/source symbol

A message with two source symbols

| | prob. | codeword |
|---|---|---|
| AA | 0.64 | 0 |
| AB | 0.16 | 10 |
| BA | 0.16 | 110 |
| BB | 0.04 | 111 |

- The Huffman Code: {0, 10, 110, 111}
- $\bar{L}$=0.64*1+ 0.16*2+0.16*3+0.04*3 = 1.56 bit/message
- 1.56 / 2 = 0.78 bit/source symbol

improvement

# Example: Coding for Binary Information Sources

| | prob. | codeword |
|---|---|---|
| AAA | 0.512 | 0 |
| AAB | 0.128 | 100 |
| ABA | 0.128 | 101 |
| ABB | 0.032 | 11100 |
| BAA | 0.128 | 110 |
| BAB | 0.032 | 11101 |
| BBA | 0.032 | 11110 |
| BBB | 0.008 | 11111 |

message with three symbols

- $\bar{L}$=0.512 × 1+ ... + 0.008 × 5 = 2.184 bit per message
- 2.184 / 3 = 0.728 bit/source symbol

| block size | $\bar{L}$ per symbol |
|---|---|
| 1 | 1.0 |
| 2 | 0.78 |
| 3 | 0.728 |
| : | : |
| | $H(S) = 0.723$ |

larger block size
⇒ more compact

# What Shannon Source Coding Theorem Means

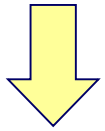
- Shannon's source coding theorem:

  The average length progressively approaches the limit H as the word size of n symbols increases

- Use block Huffman codes, and you can approach the limit.
- You never overcome the limit ($\overline{L}$ **never goes below H(X))**

Source

| | prob. |
|---|---|
| A | 0.8 |
| B | 0.2 |



$H(S) = 0.723$

| block size | ACL per symbol |
|---|---|
| 1 | 1.0 |
| 2 | 0.78 |
| 3 | 0.728 |
| : | : |
| | $0.723 + \varepsilon$ |

# Huffman Code Example

- Encode the following short text using Huffman encoding

*Eerie eyes seen near lake.*

- *The sentence has 26 characters. Their frequency of occurrence is*

| Char | Freq. | Char | Freq. | Char | Freq. |
|---|---|---|---|---|---|
| E | 1 | y | 1 | k | 1 |
| e | 8 | s | 2 | . | 1 |
| r | 2 | n | 2 | | |
| i | 1 | a | 2 | | |
| space | 4 | l | 1 | | |

- *The probability of occurrence of each character can be determined and will be used in the Huffman code.*
- *P(E) = 1/26, P(e) = 8/26, P(space) = 4/26, P(.)=1/26*

# Huffman Code Example

| Symbol | Frequency | Codeword |
|---|---|---|
| e | 8 | 11 |
| | 4 | 011 |
| n | 2 | 000 |
| r | 2 | 001 |
| s | 2 | 010 |
| a | 2 | 1001 |
| . | 1 | 10000 |
| E | 1 | 10001 |
| i | 1 | 10100 |
| k | 1 | 10101 |
| l | 1 | 10110 |
| y | 1 | 10111 |

**Summary of Results**

**H=3.16**

$\overline{L}$ **=3.23** bits/character.

Total number of bits in message =**84 bits**.

If ASCII code is used, we need 26*8= **208**

Compression:

$$\frac{84}{208} * 100\% = 40.36\%$$

Sentence: Eerie eyes seen near lake.

Code: **1000111001**…..**1010110000**

# Lempel-Ziv Code

Lecture 12

- Huffman codes have some shortcomings
  - Know symbol probability information a priori
  - Re-compute entire code if symbol probability changes
  - If source symbol probabilities are not known, one has to estimate them first.
  - Coding tree must be known at coder/decoder
  - Recall example: (***Eerie eyes seen near lake.***)
  - Based on the frequency, each symbol has a given codeword.
  - Now change to: (***Eerie Eyes Seen Near Lake.***)
  - The symbols here have different codewords. Why?
  - Capital letters have replaced small letters, thus affecting the frequency of symbols (probabilities).

# Lempel-Ziv Code

- Lempel-Ziv algorithm, named after its inventors, does not require prior knowledge of the source probabilities and uses the source output sequence itself to iteratively construct the code
- Used in *gzip*, UNIX *compress*, *LZW* algorithms
- It **is a variable to fixed length coding scheme**
- Any sequence of source symbols is uniquely parsed into phrases of **varying length**.
- Each phrase is then coded using equal length codewords

# Lempel-Ziv Code

- Works by identifying phrases of the smallest length that have not appeared so far, and maintaining these phrases in a dictionary. When a new phrase is identified, it is encoded as the concatenation of the previous phrase and the new source output.
- Number the phrases starting from 1 (0 is the empty string)
- Each phrase consists of a previously occurring phrase (**head**) followed by the new source output (**tail**).
- **Encoding:** give location of (head) followed by the additional symbol as (tail)
- Decoder uses a similar dictionary

**New phrase=previously occurring phrase + new symbol** → **Codeword=location of previously occurring phrase + new symbol**

# LZ Coding : Example 1

- Encode [abaababbbbbbbbabbbb]
- Encode [a, b, aa, ba, bb, bbb, bba, bbbb]

| Code Address | Dictionary Contents | Encoded Packets | Transmitted code |
|---|---|---|---|
| 1 | a | < 0 , a > | (000a) |
| 2 | b | < 0 , b > | (000b) |
| 3 | aa | < 1 , a > | (001a) |
| 4 | ba | < 2 , a > | (010a) |
| 5 | bb | < 2 , b > | (010b) |
| 6 | bbb | < 5 , b > | (101b) |
| 7 | bba | < 5 , a > | (101a) |
| 8 | bbbb | < 6 , b > | (110b) |

Variable length phrases

Equal length codewords

# LZ Encoding of Binary Data: Example 2

***Example** (from Proakis):*

Let us assume that we want to parse and encode the following sequence:

0100001100001010000010100000110000010100001001001 **49 Binary digits**

Parsing the sequence by the rules explained before results in the following phrases:

0, 1, 00, 001, 10, 000, 101, 0000, 01, 010, 00001, 100, 0001, 0100, 0010, 01001, … **Parsing, 16 phrases**

It is seen that all the phrases are different and each phrase is a previous phrase concatenated with a new source output. The number of phrases is 16. This means that for each phrase we need 4 bits, plus an extra bit to represent the new source output. The above sequence is encoded by

16 phrases require 4 bits (head) + Extra bit for new source symbol (tail) → 5 bits/phrase

# LZ Encoding of Binary Data: Example 2 Continued

0000 0, 0000 1, 0001 0, 0011 1, 0010 0, 0011 0, 0101 1, 0110 0,
0001 1, 1001 0, 1000 1, 0101 0, 0110 1, 1010 0, 0100 0, 1110 1, …

**Total number of bits in encoded message =16 phrases * 5 bits/phrase = 80 bits**

**Original message = 49 bits**

| Dictionary Location | | Dictionary Contents | Codeword | |
|---|---|---|---|---|
| 1 | 0001 | 0 | 0000 | 0 |
| 2 | 0010 | 1 | 0000 | 1 |
| 3 | 0011 | 00 | 0001 | 0 |
| 4 | 0100 | 001 | 0011 | 1 |
| 5 | 0101 | 10 | 0010 | 0 |
| 6 | 0110 | 000 | 0011 | 0 |
| 7 | 0111 | 101 | 0101 | 1 |
| 8 | 1000 | 0000 | 0110 | 0 |
| 9 | 1001 | 01 | 0001 | 1 |
| 10 | 1010 | 010 | 1001 | 0 |
| 11 | 1011 | 00001 | 1000 | 1 |
| 12 | 1100 | 100 | 0101 | 0 |
| 13 | 1101 | 0001 | 0110 | 1 |
| 14 | 1110 | 0010 | 1010 | 0 |
| 15 | 1111 | 0010 | 0100 | 0 |
| 16 | | | 1110 | 1 |

**Error here** ←

*** Note: 49 data bits are encoded into 80 bits**

*** Question: Where does the compression come from?**

**Answer: In short sentences, a saving can hardly be noticed. But in a long text, many phrases of longer lengths become more frequent, and as such these long phrases will be encoded into smaller number of bits.**

**0100**
**0010**
**01001**

# LZ Compression (non-binary case): Example 3

**Example 1: Use the LZ78 algorithm to encode the message**

**ABBCBCABABCAABCAAB (18 characters)**

**Solution: The encoding process is presented below in which:**

- **The symbols are parsed as:**

**A, B, BC, BCA, BA, BCAA, BCAAB**

- **We have 7 different phrases**
- **Therefore, we need three digits to represent each phrase**
- **A 1, B 2, BC 3, BCA 4**
- **BA 5, BCAA 6, BCAAB 7**
- **Encoding:**
- **<(0,A)><(0,B)><(2,C)><(3,A)><(2,A)><(4,A)><(6,B)>**

# LZ Compression (non-binary case): Example 3

- Now, we calculate the number of bits needed to represent the coded information

<(0,A)><(0,B)><(2,C)><(3,A)><(2,A)><(4,A)><(6,B)>

- We need 8 bits to represent each character using the ASCII code
- Thus, we see that the number of bits required when the string ABBCBCABABCAABCAAB: has 18 characters
- With LZ compression, the number of bits needed is:
- (3+8)+(3+8)+(3+8)+(3+8)+(3+8)+(3+8)+(3+8) = 77 bits
- With no compression, the number of binary bits is:
- (18 characters)*(8 bits/character)=144 bits.
- Efficiency = (77/144)*100=53.47%