Lecture 6:
Huffman Code

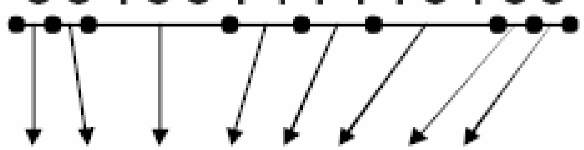
Thinh Nguyen
Oregon State University

Review

- *Coding:* Assigning binary codewords to (blocks of) source symbols.
- *Variable-length* codes (VLC)
- *Tree codes (prefix code)* are instantaneous.

Example of VLC

- Example: a 0, b 100, c 101, d 11
- Coding:
  - aabddcaa = 16 bits
  - 0 0 100 11 11 101 0 0= 14 bits
- Prefix code ensures unique decodability.
  - 00100111110100



  - a a b d d c a a

Creating a Code: The Data Compression Problem

- Assume a source with an alphabet A and known symbol probabilities $\{p_i\}$.
- Goal: Chose the codeword lengths as to minimize the bitrate, i.e., the average number of bits per symbol $\sum l_i * p_i$.
- Trivial solution: $l_i = 0 * i$.
- Restriction: We want an decodable code, so $\sum 2^{-l_i} <= 1$ (Kraft inequality) must be valid.
- Solution (at least in theory): $l_i = -\log p_i$

In practice…

- Use some nice algorithm to find the codes
  - Huffman coding
  - Tunnstall coding
  - Golomb coding

Huffman Average Code Length

- Input: Probabilities p_1, p_2, ... , p_m for symbols a_1, a_2, ... ,a_m, respectively.

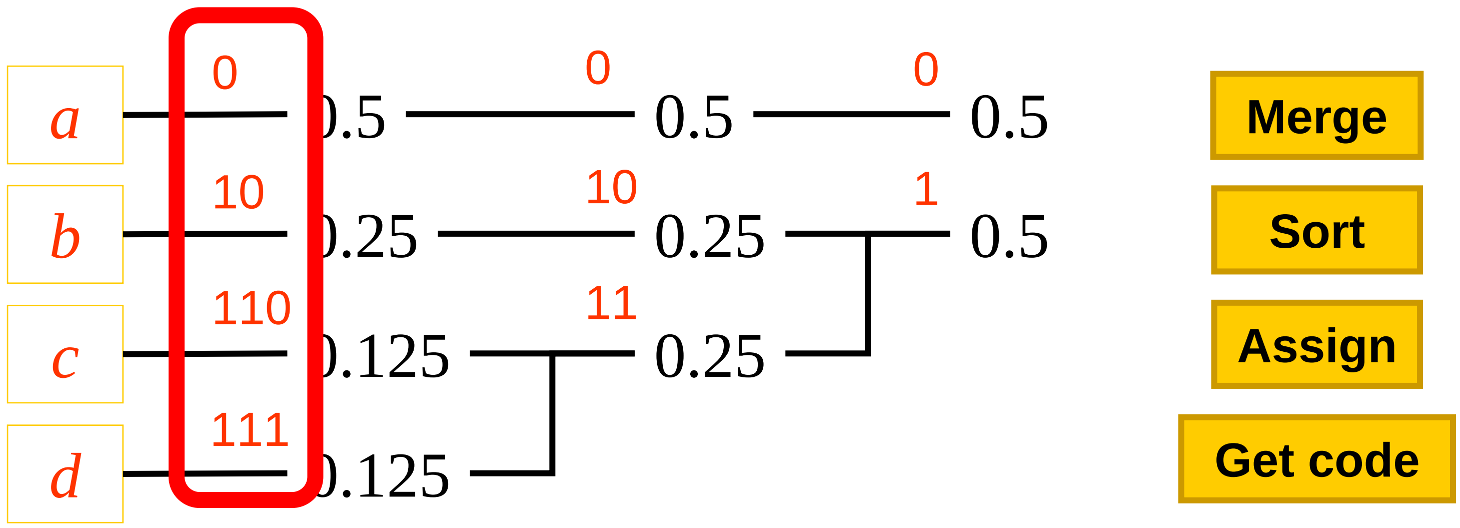
- Output: A tree that minimizes the average number of bits (bit rate) to code a symbol. That is, minimizes

$$\bar{l} = \sum_{i=1}^{m} p_i l_i$$

Where l_i is the length of codeword a_i

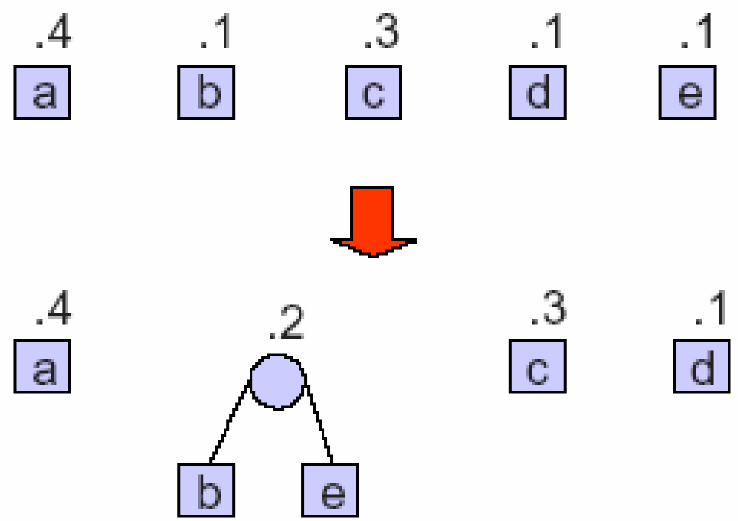
Huffman Coding

- Two-step algorithm:
  1. Iterate:
     - Merge the least probable symbols.
     - Sort.
  2. Assign bits.

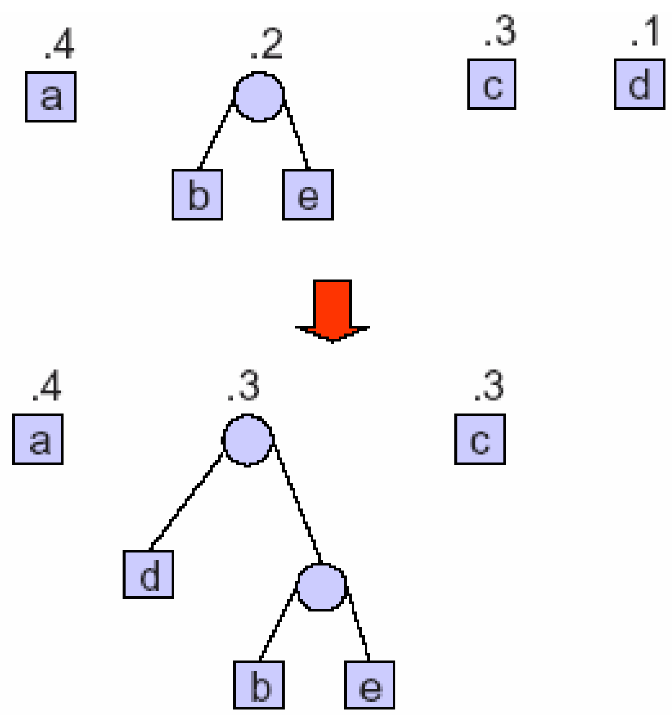


More Examples of Huffman Code

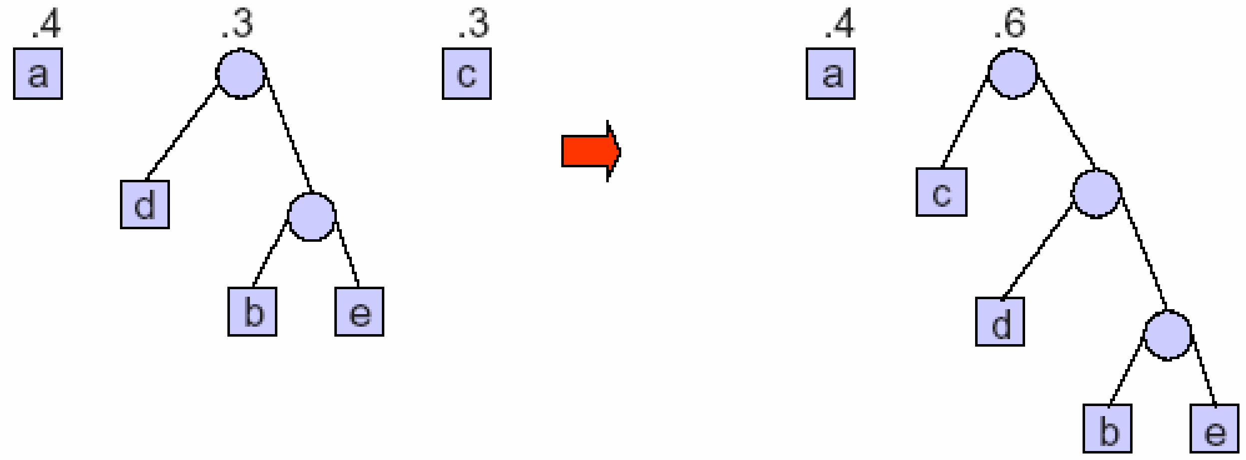
- P(a) =.4, P(b)=.1, P(c)=.3, P(d)=.1, P(e)=.1



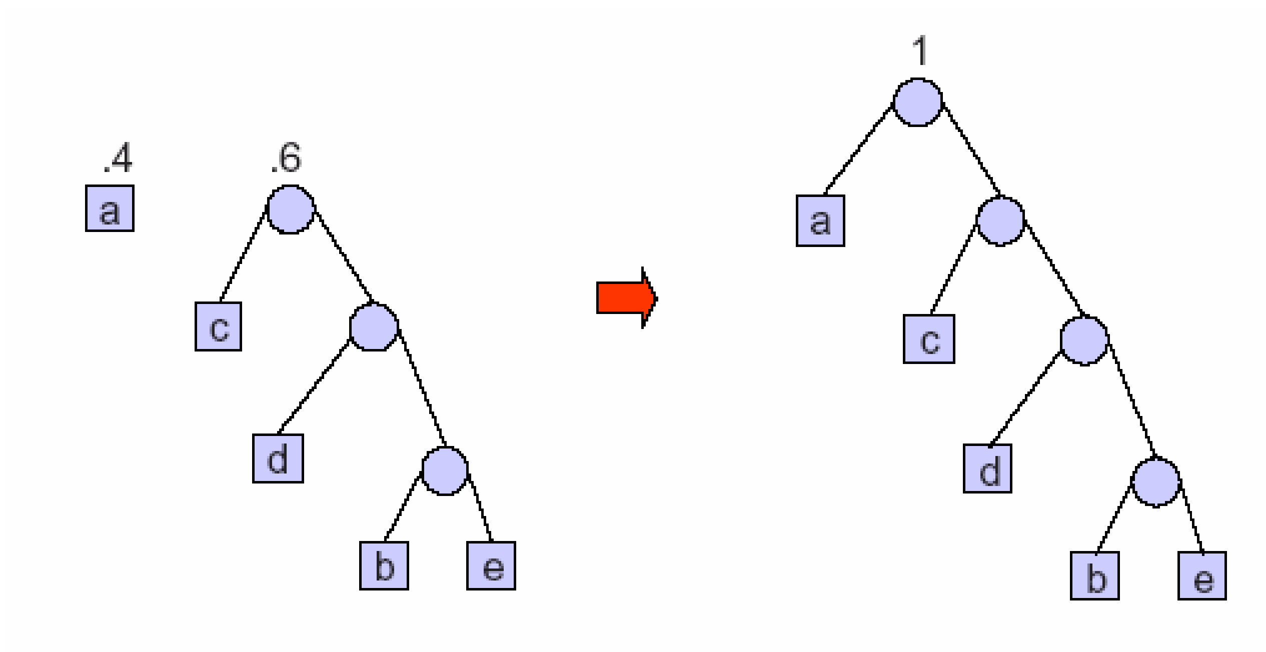
More Examples of Huffman Code



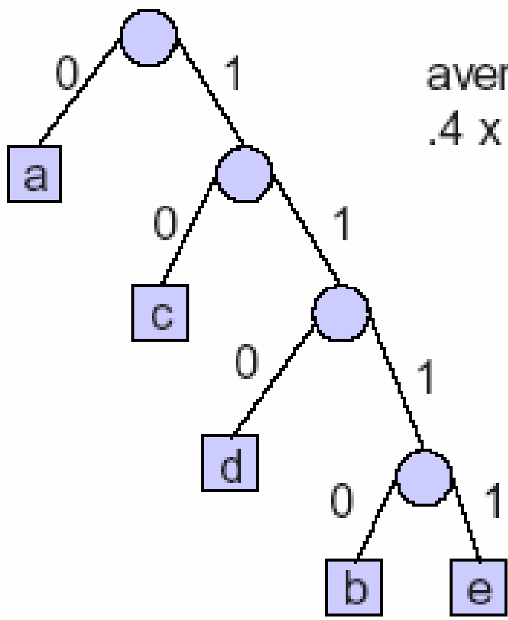
More Examples of Huffman Code



More Examples of Huffman Code



Average Huffman Code Length



average number of bits per symbol is
$.4 \times 1 + .1 \times 4 + .3 \times 2 + .1 \times 3 + .1 \times 4 = 2.1$

a	0
b	1110
c	10
d	110
e	1111

Optimality of A Prefix Code

- Necessary conditions for an optimal variable-length binary code:

1. Given any two letters a_j and a_k, if $P(a_j) >= P(a_k)$, then $l_j <= l_k$, where l_j is the length of the codeword a_j.
2. The two least probable letters have codewords with the same maximum length $l_{m.}$
3. In the tree corresponding to the optimum code, there must be two branches stemming from each intermediate node.
4. Suppose we change an intermediate node into a leaf node by combining all the leaves descending from it into a composite word of a reduced alphabet. Then if the original tree was optimal for the original alphabet, the reduced tree is optimal for the reduced alphabet.

Condition 1: If $P(a_j) >= P(a_k)$, then $l_j <= l_k$, where l_j is the length of the codeword a_j.

- Easy to see why?
- Proof by contradiction:
  - Suppose a code X is optimal with $P(a_j) >= P(a_k)$, but $l_j > l_k$
  - By simply exchanging a_j and a_k, we have a new code Y in which, its average length $= \sum l_i p_i$ is smaller than that of code X.
  - Hence, the contradition is reached. Thus, condition must hold

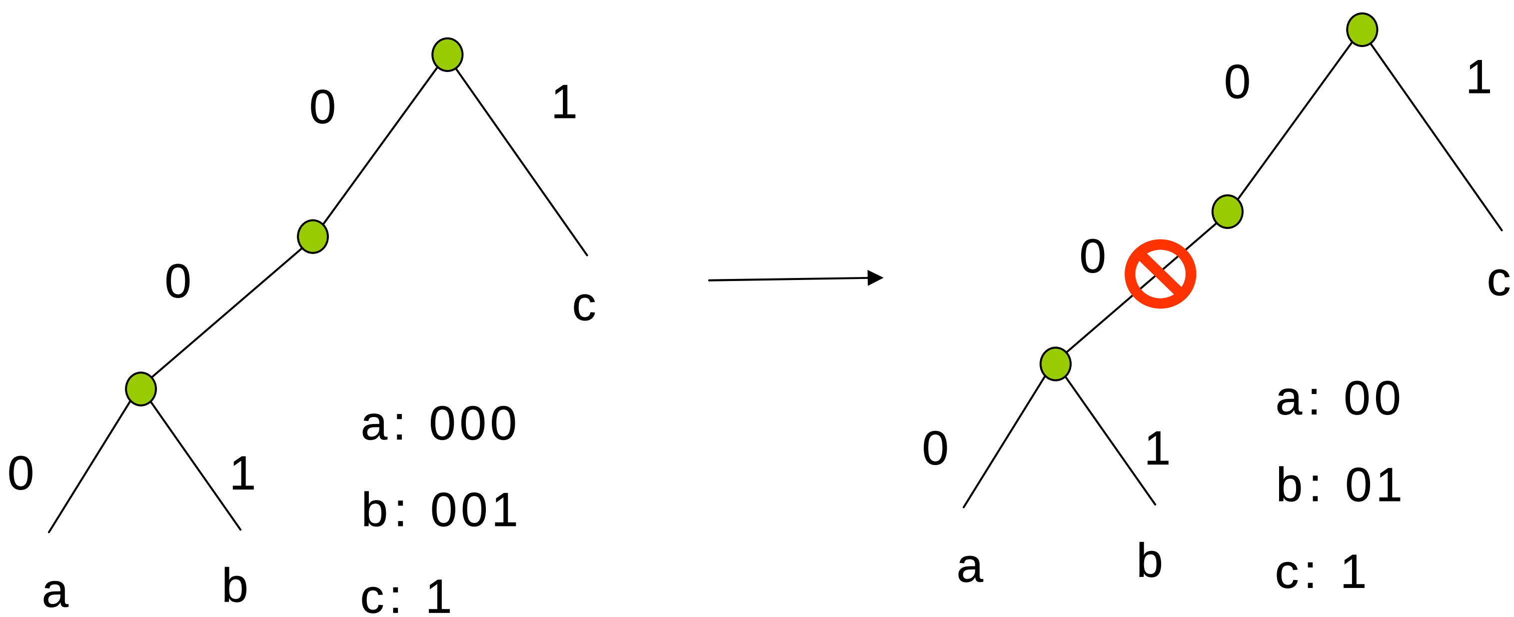
Condition 2: The two least probable letters have codewords with the same maximum length l_m.

- Easy to see why?

- Proof by contradiction:
  - Suppose we have an optimal code X in which, two codewords with lowest probabilities c_i and c_j and that c_i is longer than c_j by k bits.
  - Then because this is a prefix code, c_j cannot be the prefix to c_j. So, we can drop the last k bits of c_i.
  - We also guarantee that by dropping the last k bits of c_i, we still have a decodable codeword. This is because c_i and c_j have the longest length (least probable codes), hence they cannot be the prefix of any other code.
  - By dropping the k bits of c_i , we create a new code Y which has shorter average length, hence contradiction is reached.

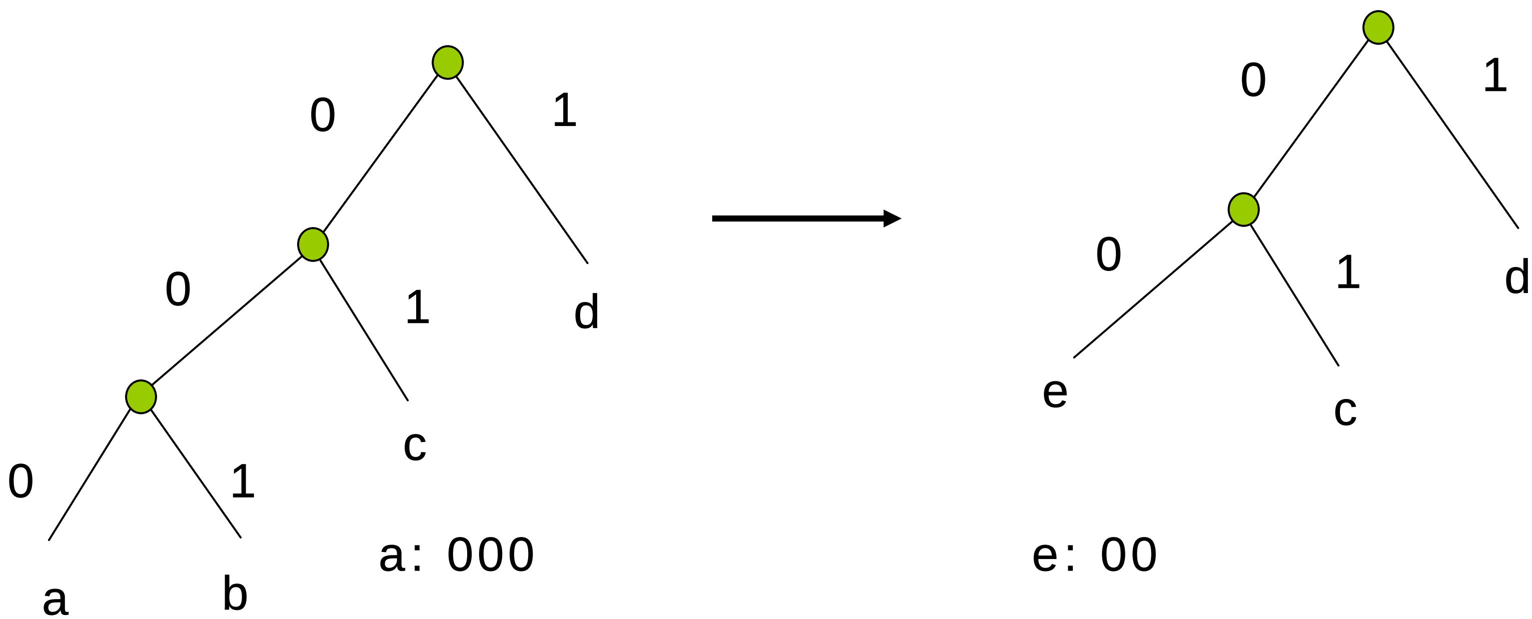
Condition 3: In the tree corresponding to the optimum code, there must be two branches stemming from each intermediate node.

- Easy to see why?
  - If there were any intermediate node with only one branch coming from that node, we could remove it without affecting the decodability of the code while reducing its average length.



Condition 4:

- Suppose we change an intermediate node into a leaf node by combining all the leaves descending from it into a composite word of a reduced alphabet. Then if the orginal tree was optimal for the original alphabet, the reduced tree is optimal for the reduced alphabet.



a: 000

b: 001

c: 01

d:1

e: 00

c: 01

d:1

Huffman code satisfies all four conditions

- Lower probable symbols are at longer depth of the tree (condition 1).
- Two lowest probable symbols have equal length (condition 2).
- Tree has two branches (condition 3).
- Code for the reduced alphabet needs to be optimum for the code of the original alphabet to be optimum by construction (condition 4)

Optimal Code Length (Huffman Code Length)

$$H(S) \le \bar{l} < H(S) + 1$$

$\bar{l}$: Average length of an optimal code

$H(S) = -\sum_{i=1}^{m} P(a_i)\log_2 P(a)_i$: Entropy of the source

Proof:

Extended Huffman Code

$$A = \{a_{1,} a_2, \ldots a_m\}, A^n = \{\underbrace{a_1 a_1 \ldots a_1}_{\text{n times}}, a_1 a_1 \ldots a_2, \ldots, a_m a_m \ldots a_m\}$$

m^n symbols in the A^{n} alphabet

$$H(S) \leq \bar{l} < H(S) + 1/n$$

$\bar{l}$: Average length of Huffman Code

$H(S)$: Entropy of the source

Proof: page 53 of the book

Huffman Coding: Pros and Cons

+ Fast implementations.

+ Error resilient: resynchronizes in $\sim l^2$ steps.

- The code tree grows exponentially when the source is extended.

- The symbol probabilities are built-in in the code.

  Hard to use Huffman coding for extended sources / large alphabets or when the symbol probabilities are varying by time.

Huffman Coding of 16-bit CD-quality audio

Filename	Original file size (bytes)	Entropy (bits)	Compressed File Size (bytes)	Compression Ratio
Mozart symphony	939,862	12.8	725,420	1.30
Folk rock (Cohn)	402,442	13.8	349,300	1.15

Huffman coding of the Differences

Filename	Original file size (bytes)	Entropy (bits)	Compressed File Size (bytes)	Compression Ratio
Mozart symphony	939,862	9.7	569,792	1.65
Folk rock (Cohn)	402,442	10.4	261,590	1.54

Complexity of Huffman Code

- O(n log(n))
  - Log(n) is the depth of the tree and n operation to compare for the lowest probabilities.

Notes on Huffman Code

- Frequencies computed for each input
  - Must transmit the Huffman code or frequencies as well as the compressed input.
  - Requires two passes
- Fixed Huffman tree designed from training data
  - Do not have to transmit the Huffman tree because it is known to the decoder.
  - H.263 video coder
- 3. Adaptive Huffman code
  - One pass
  - Huffman tree changes as frequencies change