UNIT 6B

Data Representation: Exploiting Redundancy

Last Lecture

- Encoding unsigned and signed integers
- Encoding Characters as Integers, Ascii Table

This lecture

- Parity: injecting redundancy for error detection
- Redundancy in information
- Data compression
- Removing redundancy for data compression
  - Huffman codes

error correction using

PARITY BITS

Noisy Communication Channels

- Suppose we're sending ASCII characters over the network
- Network communications may erroneously alter bits of a message
- Simple error detection method: **the parity bit**

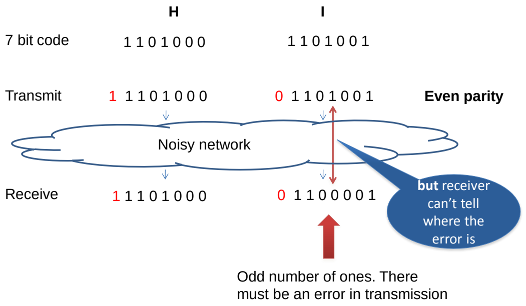
Parity

- Idea: for each character (sequence of 7 bits), count the number of bits that are 1
- Sender and receiver agree to use *even parity* or *odd parity*; sender sends **extra** leftmost bit
  - Even parity: Set the leftmost bit so that the number of 1's in the byte is even.
  - Odd parity: Set the leftmost bit so that the number of 1's in the byte is odd.

Parity Example

- "M" is transmitted using **even parity**.
- "M" in ASCII is 77_{10}, or 100 1101 in binary
  - four of these bits are 1
- Transmit 0 100 1101 to make the number of 1-bits **even**.
- Receiver counts the number of 1-bits in character received
  - if odd, something went wrong, request retransmission
  - if even, proceed normally
  - Two bits could have been flipped, giving the illusion of correctness. **But** the probability of 2 or more bits in error is low.

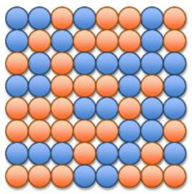
Parity Example



	H	**I**	
7 bit code	1 1 0 1 0 0 0	1 1 0 1 0 0 1	
Transmit	1 1 1 0 1 0 0 0	0 1 1 0 1 0 0 1	**Even parity**
Receive	1 1 1 0 1 0 0 0	0 1 1 0 0 0 0 1	

but receiver can't tell where the error is

Odd number of ones. There must be an error in transmission



🟦	🟦	🟧	🟦	🟧	🟧	🟦	🟧
🟦	🟧	🟦	🟧	🟧	🟧	🟦	🟧
🟧	🟦	🟧	🟦	🟦	🟦	🟦	🟦
🟦	🟦	🟧	🟧	🟧	🟦	🟧	🟦
🟧	🟧	🟧	🟦	🟦	🟧	🟦	🟦
🟧	🟧	🟧	🟧	🟧	🟦	🟦	🟧
🟦	🟦	🟦	🟧	🟦	🟦	🟧	🟦
🟧	🟧	🟧	🟦	🟦	🟧	🟧	🟧

- Seven characters are transmitted here as bytes using even parity along with a special 8th byte.
- The two colors represent 1's and 0's.
- One bit is in error. Can you find it?

Parity and redundancy

- An ASCII character with a correct parity bit contains *redundant information*
- …because the parity bit is *predictable* from the other bits
- This idea leads into the basics of information theory

a powerful tool

REDUNDANT INFORMATION

Information Content

- We measure information content in bits
  - This is related to the fact that we can represent 2^k different things with k bits.
  - Turn the idea around and if we want to represent M different things, we need $\log_2 M$ bits
- **But** this is only true if the M things all have the same probability

Probability and information content

When you get an item of information, how surprised are you? For example, your phone tells you that you have a text. Who from?

•your best friend: you’re not surprised; this event has *high probability*

•Barack Obama: you’re surprised; this event has *low probability*

Probability and information content

- **Low probability** events have **high** information content; when you learn of them you get a lot of new information
  - *Barack Obama knows my phone number!!!!*
- **High probability** events have **low** information content.
  - *The sun rose in the east this morning. meh*
- Notice that a character with correct parity is much more probable than one with incorrect parity

squeezing out redundancy

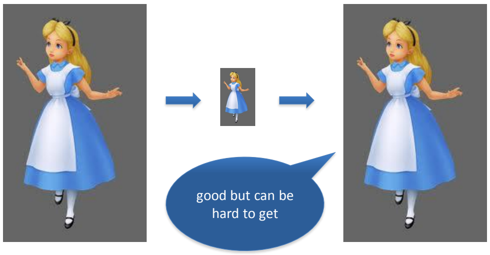
DATA COMPRESSION

Data Compression: Why?

- Faster transmission
  - e.g. digital video impossible without compression
- Cheaper storage
  - e.g. OS X Mavericks compresses data in memory until it needs to be used

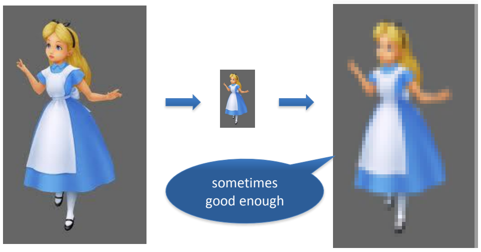
Data Compression: choices

- **Lossless compression**



Data Compression: choices

- **Lossy compression**



Today: lossless text compression

- Compression:
  - Input: fixed-width character codes (e.g. 7-bit ASCII codes)
  - Output: Huffman codes (variable number of bits per character)
- Decompression:
  - Huffman codes to fixed-length codes
- Idea: squeeze out redundancy indicated by character probabilities

ASCII: Fixed-Width Encoding

- Remember: each character is given a binary code with 7 bits.
- This gives us 2^7 = 128 different codes for characters.
- Can we make do with fewer bits? Suppose our text is entirely in Hawaiian…

The Hawaiian Alphabet

- The Hawaiian alphabet consists of 13 characters.
  - **'** is the okina which sometimes occurs between vowels (e.g. **KAMA'AINA**)

'
A
E
H
I
K
L
M
N
O
P
U
W

Specialized fixed-width encodings

- Suppose our text file is entirely in Hawaiian

- How many bits do we need for our 13 characters?
  - Are 3 bits enough? 000, 001, …, 111?
  - Are 4 bits enough? 0000, 0001, …, 1111?
  - In general, for *k equally probable* characters we need $\lceil \log_2 k \rceil$ bits

- So for Hawaiian we need $\lceil \log_2 13 \rceil = 4$ bits

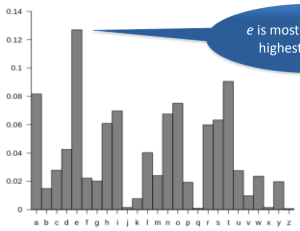
Cost of Fixed-Width Encoding

- With a fixed-width encoding scheme of *n* bits and a file with *m* characters, need *mn* bits to store the entire file.
  - Example: to store 1000 characters of Hawaiian we would need 4000 bits
- Can we do better? **Idea:** some characters are used much more often than others.
  - If we assign fewer bits to more frequent characters, and more bits to less frequent characters, then the overall length of the message might be shorter.

Use a method known as Huffman encoding named after David Huffman

Frequency counts as probabilities

- **Example:** counting the relative frequency of letters in a large corpus of English text



e is most frequent, has highest probability

image: Wikipedia

Hawaiian Alphabet Frequencies

- The table to the right shows each character along with its relative frequency in Hawaiian words.
- Smaller numbers mean less common characters
- Frequencies add up to 1.00 and can be viewed as *probabilities*

'	0.068
A	0.262
E	0.072
H	0.045
I	0.084
K	0.106
L	0.044
M	0.032
N	0.083
O	0.106
P	0.030
U	0.059
W	0.009

Huffman Coding: the process

1. Assign character codes
   a. Obtain character frequencies
   b. Use frequencies to build a *Huffman tree*
   c. Use tree to assign variable-length codes to characters (store them in a table)
2. Use table to encode (compress) ASCII source file to variable-length codes
3. Use tree to decode (decompress) to ASCII

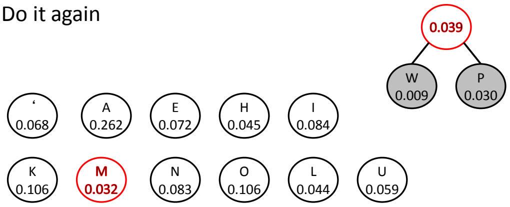
Building The Huffman Tree

- We use a tree structure to develop the unique binary code for each letter.
- Start with each letter/frequency as its own single-node tree
- Find the **two lowest-frequency** trees

‘ 0.068
A 0.262
E 0.072
H 0.045
I 0.084
K 0.106
L 0.044
M 0.032
N 0.083
O 0.106
P 0.030
U 0.059
W 0.009

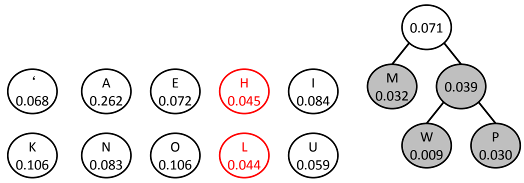
Building The Huffman Tree

- Combine **two lowest-frequency** trees into a tree with a new root with the sum of their frequencies.
- Do it again

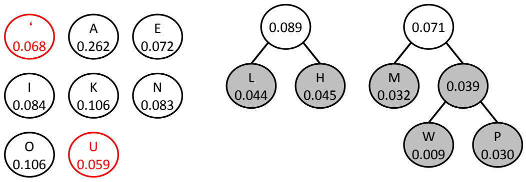


Building The Huffman Tree

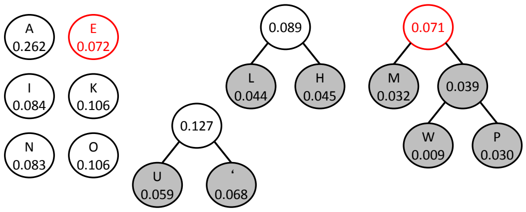
- …and again, as many times as possible



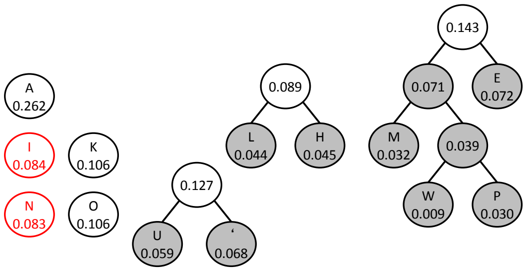
Building The Huffman Tree



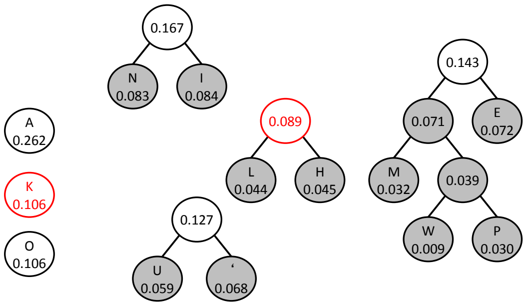
Building The Huffman Tree



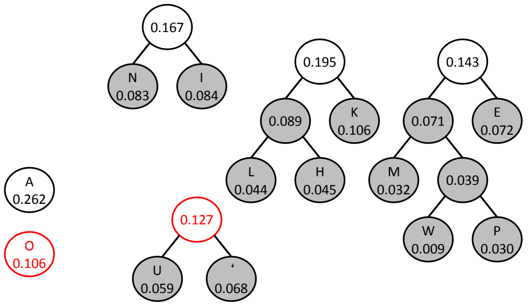
Building The Huffman Tree



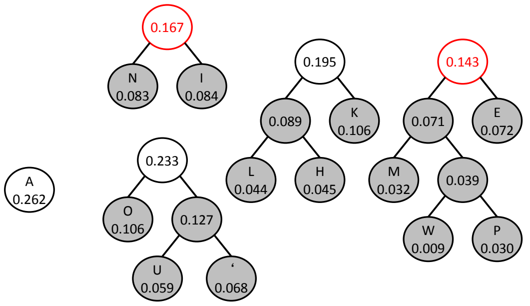
Building The Huffman Tree



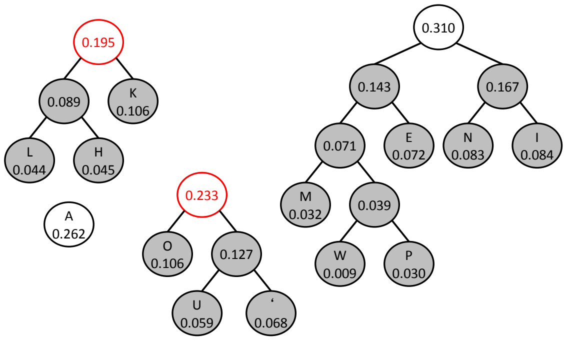
Building The Huffman Tree



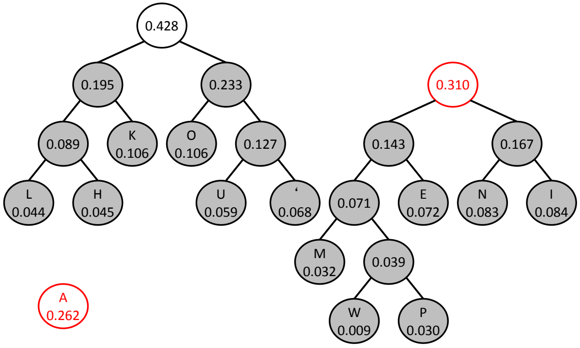
Building The Huffman Tree



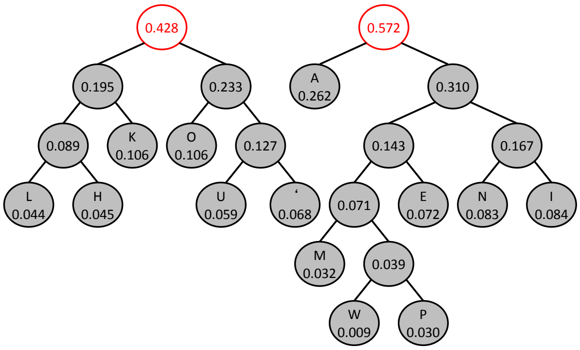
Building The Huffman Tree

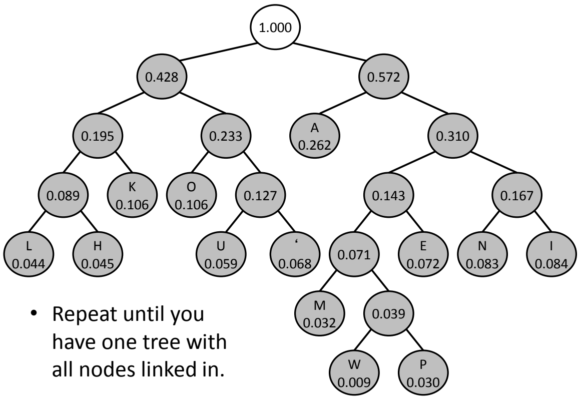


Building The Huffman Tree



Building The Huffman Tree

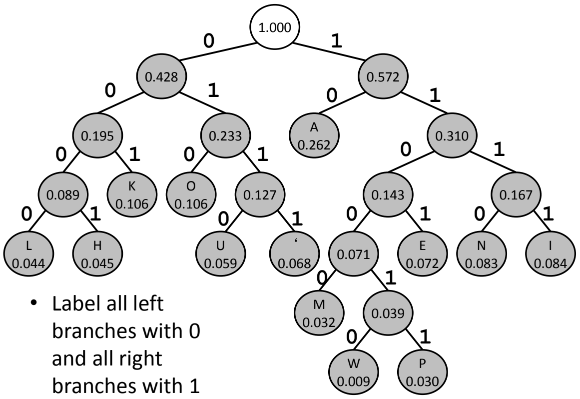




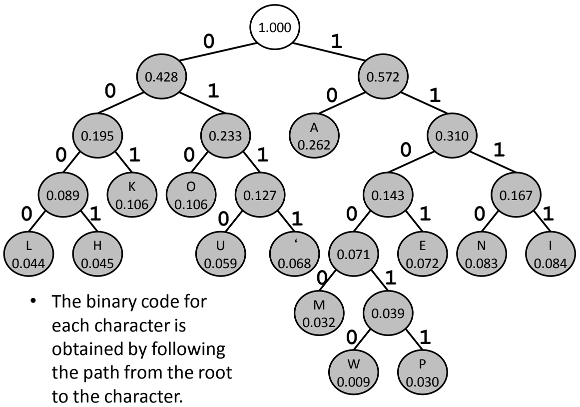
- Repeat until you have one tree with all nodes linked in.

Using the Tree to Assign Codes

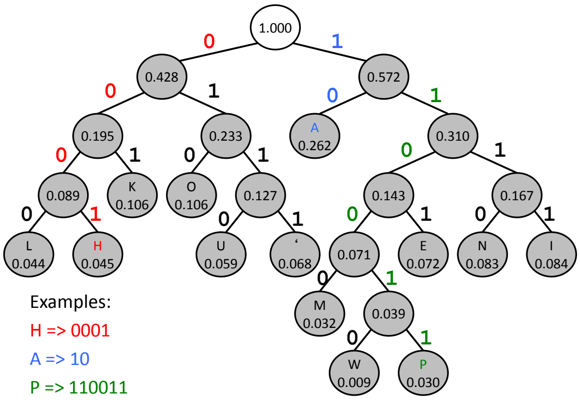
- The path from the root to each character determines the code



- Label all left branches with 0 and all right branches with 1



- The binary code for each character is obtained by following the path from the root to the character.



Examples:

H => 0001

A => 10

P => 110011

Fixed Width vs. Huffman Coding

'	0000	'	0111
A	0001	A	10
E	0010	E	1101
H	0011	H	0001
I	0100	I	1111
K	0101	K	001
L	0110	L	0000
M	0111	M	11000
N	1000	N	1110
O	1001	O	010
P	1010	P	110011
U	1011	U	0110
W	1100	W	110010

ALOHA

Fixed Width:
0001 0110 1001 0011 0001
20 bits

Huffman Code:
10 0000 010 0001 10
15 bits

How about…

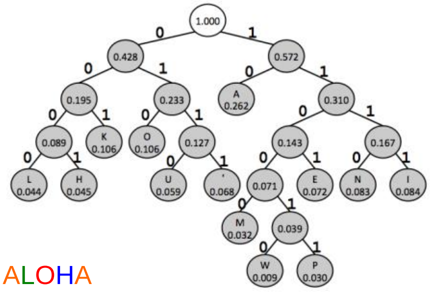
- `humuhumunukunukuapua'a` (22 chars) (the reef triggerfish)
- `4454445444344434264242 = 84`
- `vs 22*4 = 88`

Decoding

- In a fixed-width code, the boundaries between letters are fixed in advance:
  `0001 0110 1001 0011 0001`
- With Huffman codes, the boundaries are determined by the letters themselves.
  - No letter's code can be a prefix of another letter.
  - Example: since A is "10", no other letter's code can begin with "10". All the remaining codes begin with "00", "01", or "11".

Decoding

100000010000110

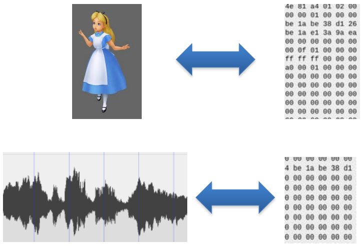


ALOHA

- To find the character use the bits to determine path from root

Next

- Representing images and sound



```
4e 81 a4 01 02 00
00 00 01 00 00 00
be 1a be 38 d1 26
be 1a e1 3a 9a ea
00 00 00 00 00 00
00 0f 01 00 00 00
ff ff ff 00 00 00
a0 00 01 00 00 00
00 00 00 00 00 00
00 00 00 00 00 00
00 00 00 00 00 00
00 00 00 00 00 00
00 00 00 00 00 00
```

```
4 be 1a be 38 d1
0 00 00 00 00 00
0 00 00 00 00 00
0 00 00 00 00 00
0 00 00 00 00 00
0 00 00 00 00 00
0 00 00 00 00 00
0 00 00 00 00 00
```