6.02
Spring 2011

[Home](#)
[Announcements](#)

[Handouts](#)
»[Lectures](#)
»[PSets](#)
»[Tutorial Problems](#)

*MIT cert required
* [On-line grades](#)
* PSets:
[1](#),
* [Help queue](#)
* [Lab Hours](#)
* [Staff only](#)

[Course info](#)
[Course calendar](#)
[Course description](#)

[Python](#)
[Numpy](#)
[Matplotlib](#)

[Previous terms](#)

6.02 Course Information

Prerequisites 8.02, 18.03, 6.01.
The labs require familiarity with Python.

Units 4-4-4
Requirements satisfied: 1/2 Institute Lab, 6 Engineering Design Points.

Lectures MW 2-3 pm in 34-101.

Recitations

#	Time	Room	Instructor
1	TR 10	24-402	Shah
2	TR 11	24-402	Shah
3	TR 12	38-166	Lim
4	TR 1	38-166	Sun
5	TR 2	38-166	Sun

Help The course staff has office hours in the afternoons and evenings in the 6.02 lab, 38-530. The staffing schedule is posted on the Lab Hours page on the course website. The lab has 100 debathena workstations (or bring your own laptop); hours are posted below.

Hours	Days
0900 - 2330	Mon - Thu
0900 - 1700	Fri
closed	Sat
1300 - 2330	Sun

There are special hours during holidays and breaks -- see the schedule posted in the lab for more details.

You can also try email to 6.02-help at mit dot edu, although it's hard to debug Python code via email :)

If you are having access or technical problems with the on-line system, please email 6.02-web at mit dot edu.

Staff

Duties	Name	Email at mit.edu	Office	Phone
Lectures	Chris Terman	cjt	32-G790	x3-6038
Recitations	Fabian Lim	flim	38-266	x4-4913
	Devavrat Shah	devavrat	32-D670	x3-4670
	John Sun	johnsun	36-680D	x4-5287
TAs	Sidhant Misra	sidhant	--	--
	Chen Sun	sunchen	--	--
	Xiawa Wang	xiawaw	--	--
	Grace Woo	gracewoo	--	--

Weekly PSets There are weekly on-line problem sets, posted on the website most Wednesdays, which are due at 6am the following Thursday morning. Solutions will be available after the due date once you have submitted the assignment on-line.

Some of the problems will involve writing Python functions, so be sure to leave time to debug your implementation before the due date. There is a 10-minute checkoff interview each week which must be scheduled with your assigned staff member within five days after the assignment is due (i.e., by the end of Monday).

After grading, your score and any comments from the grader can be viewed on-line by browsing the pset.

You must complete the interview for each pset as a prerequisite for passing the course. A missing interview will result in a failing grade; incompletes will *not* be given for missing interviews.

Late policy: Your grade will be multiplied by 0.5 for late submissions. Late submissions will be accepted for 5 days after the due date. You can extend the submission deadline by 5 days, avoiding the late penalty, for up to two psets during the term -- see your On-lines Grades page. Note that an extension eliminates the late penalty but doesn't change the 5-day deadline.

If you have a note from Student Support Services, please see your TA. For all other circumstances (interview trips, sporting events, performances, overwork, etc.) you may use your extensions.

Collaboration policy: The assignments are intended to help you understand the material and should be done individually. You're welcome to get help from other students and the course staff but **the work you hand in must be your own.** Copying another person's work or allowing your work to be copied by others is a serious academic offense and will be treated as such. We do spot-check your submissions for infractions of the collaboration policy so please don't tempt fate by submitting someone else's work as your own; it will save us all a lot of grief.

Quizzes

There are three quizzes, scheduled as follows:

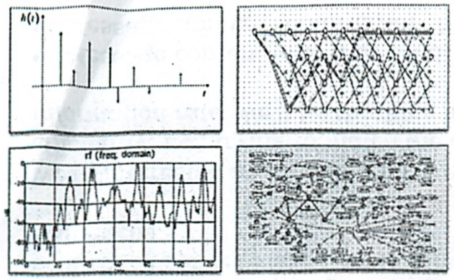
Quiz 1: March 3, 2011, 7:30-9:30, Room 50-340
Quiz 2: April 12, 2011, 7:30-9:30, Room 26-100
Quiz 3: During final exams week, not yet scheduled

Grading

Your final grade will be determined by a weighted average of the following:

Quiz 1: 16%
Quiz 2: 17%
Quiz 3: 17%
10 PSets: 5% each, for a total of 50%

To review your current scores use the "On-line grades" link in the nav bar to the left.



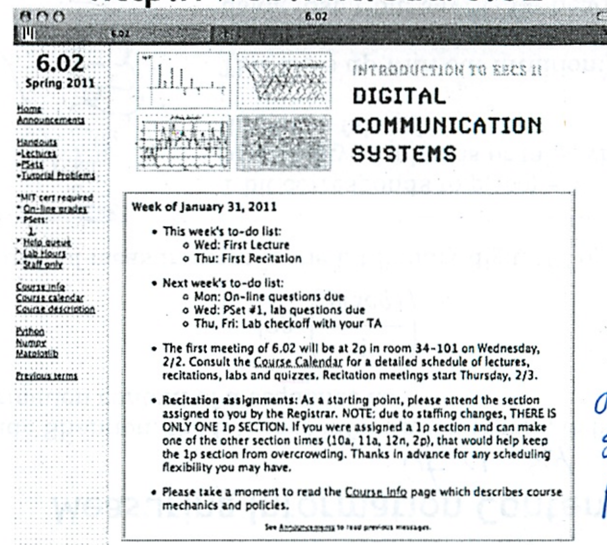
INTRODUCTION TO EECS II

DIGITAL COMMUNICATION SYSTEMS

6.02 Spring 2011
Lecture #1

- Engineering goals for communication systems
- Measuring information
- Huffman codes

http://web.mit.edu/6.02



6.02
Spring 2011

Home
Announcements
Handouts
Lectures
PSets
Tutorial Problems

*MIT cert required
On-line grades
PSets:
Help queue
Lab Hours
Staff only

Course info
Course calendar
Course description

Python
Numpy
Matplotlib

Previous terms

INTRODUCTION TO EECS II
DIGITAL COMMUNICATION SYSTEMS

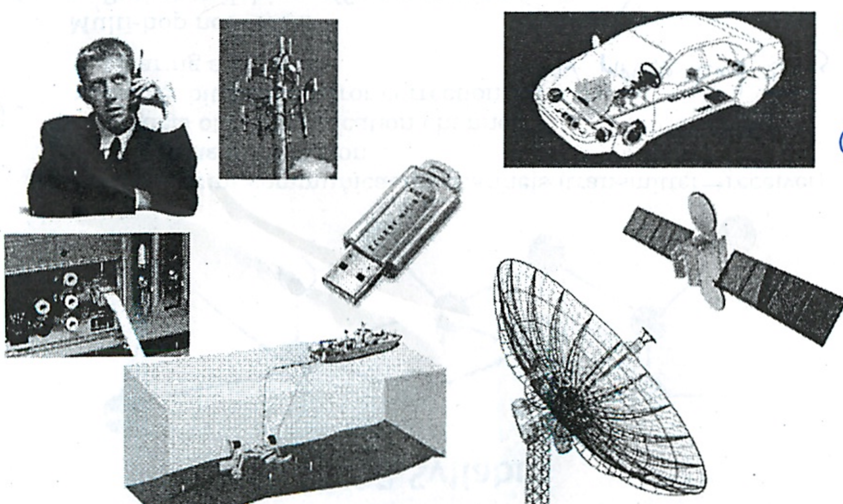
Week of January 31, 2011

- This week's to-do list:
  - Wed: First Lecture
  - Thu: First Recitation
- Next week's to-do list:
  - Mon: On-line questions due
  - Wed: PSet #1, lab questions due
  - Thu, Fri: Lab checkoff with your TA
- The first meeting of 6.02 will be at 2p in room 34-101 on Wednesday, 2/2. Consult the Course Calendar for a detailed schedule of lectures, recitations, labs and quizzes. Recitation meetings start Thursday, 2/3.
- **Recitation assignments:** As a starting point, please attend the section assigned to you by the Registrar. NOTE: due to staffing changes, THERE IS ONLY ONE 1p SECTION. If you were assigned a 1p section and can make one of the other section times (10a, 11a, 12n, 2p), that would help keep the 1p section from overcrowding. Thanks in advance for any scheduling flexibility you may have.
- Please take a moment to read the Course Info page which describes course mechanics and policies.

See Announcements to read previous messages.

announcements
scores
P-sets

Digital Communications



transmitter
recieve/

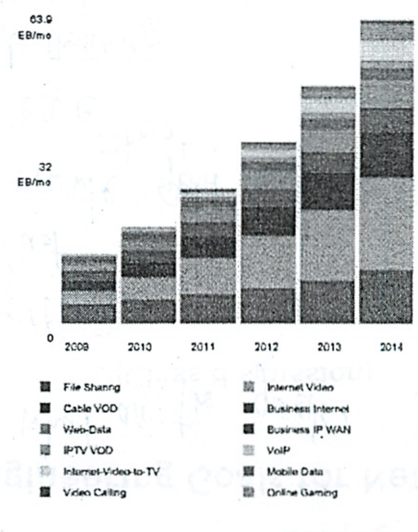
convert everything
to 1, 0
and then transmit
those reliably

modeling
is important

storage = communication across time

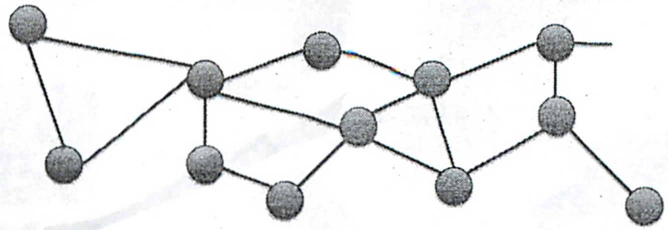
2^{70}

Internet: 10^{21} bytes/year by 2014!



*Cisco VNI June 2010

6.02 Syllabus



Point-to-point communication channels (transmitter→receiver):
- Encoding information
- Models of communication channels
- Noise, bit errors, error correction
- Sharing a channel

Multi-hop networks:
- Packet switching, efficient routing
- Reliable delivery on top of a best-efforts network

how modern hard drives work

*building reliable system in imperfect world

Engineering Goals for Networks

what are the goals?

(Class discussion)

efficient
fast
reliable - spend most of time on
- robust
secure
authenticity
mobile
low-cost
scalable - also spending time on

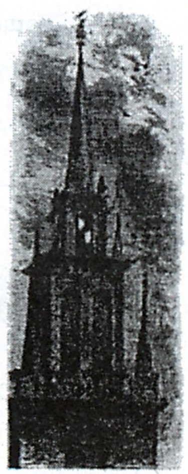
Information Resolves Uncertainty

In information theory, information is a mathematical quantity expressing the probability of occurrence of a particular sequence of symbols as contrasted with that of alternative sequences.

Information content of a sequence *increases* as the probability of the sequence *decreases* – likely sequences convey less information than unlikely sequences.

We're interested in encoding information efficiently, i.e., trying to *match the data rate to the information rate.* We'll be thinking about:

- Message content (one if by land, two if by sea)
- Message timing (No lanterns? No message!)



How the reciever know that it is recieving a message?

Measure info content in message

other stuff to worry about

Measuring Information Content

of a seq

Claude Shannon, the father of information theory, defined the information content of a sequence as

$$\log_2\left(\frac{1}{p(seq)}\right)$$

The unit of measurement is the bit (binary digit: "0" or "1").

1 bit corresponds to p(seq) = ½, e.g., the probability of a heads or tails when flipping a fair coin.

This lines up with our intuition: we can encode the result of a single coin flip using just 1 bit: say "1" for heads, "0" for tails. Encoding 10 flips requires 10 bits.

seq $= \frac{1}{2}$ → get 1

Very hard to come up w/ encoding to match info content

Expected Information Content: Entropy

Now consider a message transmitting the outcome of an event that has a set of possible outcomes, where we know the probability of each outcome.

(How much info do I need to send you to transmate outcome

Mathematicians would model the event using a discrete random variable X with possible values $\{x_1, \ldots, x_n\}$ and their associated probabilities $p(x_1), \ldots, p(x_n)$.

The *entropy* H of a discrete random variable X is the expected value of the information content of X:

$$H(X) = E(I(X)) = \sum_{i=1}^{n} p(x_i)\log_2\left(\frac{1}{p(x_i)}\right)$$

Computing weighted avg of info content

↳ called entropy

know prob of the outcomes

Okay, why do we care about entropy?

Entropy tells us the average amount of information that must be delivered in order to resolve all uncertainty. This is a lower bound on the number of bits that must, on the average, be used to encode our messages.

If we send fewer bits on average, the receiver will have some uncertainty about the outcome described by the message.

If we send more bits on average, we're "wasting" the capacity of the communications channel by sending bits we don't have to. "Wasting" is in quotes because, alas, it's not always possible to find an encoding where the data rate matches the information rate.

Achieving the entropy bound is the "gold standard" for an encoding: entropy gives us a metric to measure encoding effectiveness.

total # possibilities = $2^{10} = 1024$

note: its the avg amt of bits sent - not just best case message

Special case: all p_i are equal

Suppose we're in communication about an event where all N outcomes are equally probable, i.e., $p(x_i) = 1/N$ for all i. Uniform

$$H_{before}(X) = \sum_{i=1}^{N}\left(\frac{1}{N}\right)\log_2\left(\frac{1}{1/N}\right) = \log_2(N)$$

If you receive a message that reduces the set of possible outcomes to M equally probable choices, the entropy after the receipt of the message is

$$H_{after}(X) = \sum_{i=1}^{M}\left(\frac{1}{M}\right)\log_2\left(\frac{1}{1/M}\right) = \log_2(M)$$

what is it you still don't know

The information content of the received message is given by the change in entropy:

$$H_{message}(X) = H_{before} - H_{after} = \log_2(N) - \log_2(M) = \boxed{\log_2(N/M)}$$

how much entropy decreased

how much info was in message

Example

We're drawing cards at random from a standard 52-card deck:

Q. If I tell you the card is a ♤, how many bits of information have you received?

A. We've gone from N=52 possible cards down to M=13 possible cards, so the amount of info received is $\log_2(52/13) = 2$ bits.

This makes sense, we can encode one of the 4 (equally probable) suits using 2 bits, e.g., 00=♡, 01=♢, 10=♧, 11=♤.

Q. If instead I tell you the card is a seven, how much info?

A. N=52, M=4, so info = $\log_2(52/4) = \log_2(13) = 3.7$ bits

Hmm, what does it mean to have a fractional bit? not directly related to a possible encoding

some inefficency

Example (cont'd.)

Q. If I tell you the card is the 7 of spades, how many bits of information have you received?

A. We've gone from N=52 possible cards down to M=1 possible cards, so the amount of info received is $\log_2(52/1) = 5.7$ bits.

Note that information is additive (5.7 = 3 + 2.7)!

But this is true only when the separate pieces of information are independent (not redundant in any way).

So if I sent first sent a message the card was black (i.e., a ♤ or ♧) – 1 bit of information since p(♤ or ♧) = ½ – and then sent the message it was a spade, the total information received is *not* the sum of the information content of the two messages since the information in the second message overlaps the information of the first message.

Fixed-length Encodings

An obvious choice for encoding equally probable outcomes is to choose a fixed-length code that has enough sequences to encode the necessary information

- 96 printing characters → 7-bit ASCII
- Unicode characters → UTF-16
- 10 decimal digits → 4-bit BCD (binary coded decimal)

Fixed-length codes have some advantages:

- They are "random access" in the sense that to decode the n^{th} message symbol one can decode the n^{th} fixed-length sequence without decoding sequence 1 through n-1.
- Table lookup suffices for encoding and decoding

Improving on Fixed-length Encodings

$choice_i$	p_i	$\log_2(1/p_i)$
"A"	1/3	1.58 bits
"B"	1/2	1 bit
"C"	1/12	3.58 bits
"D"	1/12	3.58 bits

Variable length

The expected information content in a choice is given by the entropy:
= (.333)(1.58) + (.5)(1) + (2)(.083)(3.58) = 1.626 bits

Can we find an encoding where transmitting 1000 choices requires 1626 bits on the average?

The "natural" fixed-length encoding uses two bits for each choice, so transmitting the results of 1000 choices requires 2000 bits.

need to start at begining

Variable-length encodings

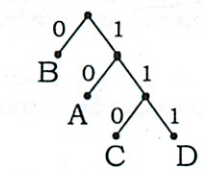
(David Huffman, MIT 1950)

Use shorter bit sequences for high probability choices, longer sequences for less probable choices

$choice_i$	p_i	encoding
"A"	1/3	10
"B"	1/2	0
"C"	1/12	110
"D"	1/12	111

BC A BA D

011010010111



Huffman Decoding Tree

Expected length
=(.333)(2)+(.5)(1)+(2)(.083)(3)
= 1.666 bits

Transmitting 1000 choices takes an average of 1666 bits... better but not optimal

Variable length coding – more frequent get shorter encoding

resolves issue when knowing where to stop "prefix coding"

BAD DAD
010111 ----

Another Variable-length Code (not!)

Here's an alternative variable-length for the example on the previous page:

Letter	*Encoding*
A	0
B	1
C	00
D	01

Why isn't this a workable code?

The expected length of an encoded message is

$$(.333+.5)(1) + (.083 + .083)(2) = 1.22 \text{ bits}$$

AA = C
AB = D

which even beats the entropy bound ☺

don't know where to stop in reading bits

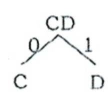
recitation

Huffman's Coding Algorithm

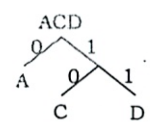
- Begin with the set S of symbols to be encoded as binary strings, together with the probability p(s) for each symbol s in S. The probabilities sum to 1 and measure the frequencies with which each symbol appears in the input stream. In the example from the previous slide, the initial set S contains the four symbols and their associated probabilities from the table.
- Repeat the following steps until there is only 1 symbol left in S:
  - Choose the two members of S having lowest probabilities. Choose arbitrarily to resolve ties.
  - Remove the selected symbols from S, and create a new node of the decoding tree whose children (sub-nodes) are the symbols you've removed. Label the left branch with a "0", and the right branch with a "1".
  - Add to S a new symbol that represents this new node. Assign this new symbol a probability equal to the sum of the probabilities of the two nodes it replaces.

Huffman Coding Example

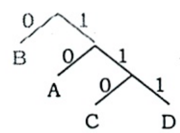
- Initially S = { (A, 1/3) (B, 1/2) (C, 1/12) (D, 1/12) }
- First iteration
  - Symbols in S with lowest probabilities: C and D
  - Create new node
  - Add new symbol to S = { (A, 1/3) (B, 1/2) (CD, 1/6) }



- Second iteration
  - Symbols in S with lowest probabilities: A and CD
  - Create new node
  - Add new symbol to S = { (B, 1/2) (ACD, 1/2) }



- Third iteration
  - Symbols in S with lowest probabilities: B and ACD
  - Create new node
  - Add new symbol to S = { (BACD, 1) }



- Done

Huffman Codes - the final word?

- Given static symbol probabilities, the Huffman algorithm creates an optimal encoding when each symbol is encoded separately. (optimal ≡ no other encoding will have a shorter expected message length)
- Huffman codes have the biggest impact on average message length when some symbols are substantially more likely than other symbols.
- You can improve the results by adding encodings for symbol pairs, triples, quads, etc. From example code:
  Pairs: 1.646 bits/sym, Triples: 1.637, Quads 1.633, ...
  But the number of possible encodings quickly becomes intractable.
- Symbol probabilities change message-to-message, or even within a single message.
- Can we do adaptive variable-length encoding?
  - Tune in next time!

build from bottom up

P-Set 1 posted

6.02 Recitation 1

2/3

Devavrat Shah 32-D670

What does information mean?

Measured in bits → 1 or 0

Information "measure"

~~Have to~~

Reduces uncertainty ~~of~~

If info tells you something you already know

↳ its not information

Content of information

$$p \log\left(\frac{1}{p}\right)$$

Say 4 possible grades

A 00

B 01

C 10

D 11

But what if prob not equal?

If savings is greater than loss from extra bits

(2)

So if

$\frac{1}{2}$ A

$\frac{1}{6}$ B

$\frac{1}{6}$ C

$\frac{1}{6}$ D

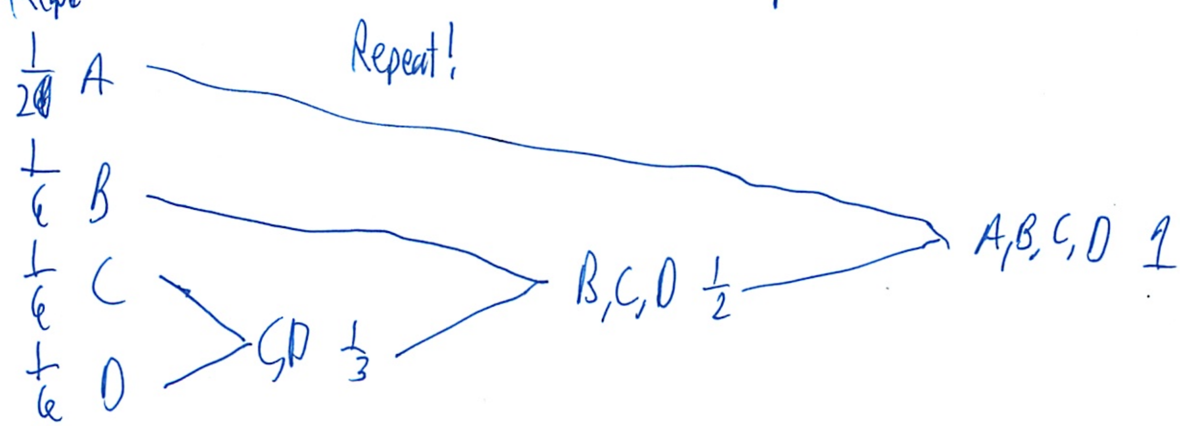
Then $H_{er} = \frac{1}{2}\log_2 2 + \frac{1}{6}\log_2 6 + \frac{1}{6}\log 6 + \frac{1}{6}\log 6$

This is the theoritical limit $\approx \frac{7}{4}$

The question is can we achieve it

Huffman Coding

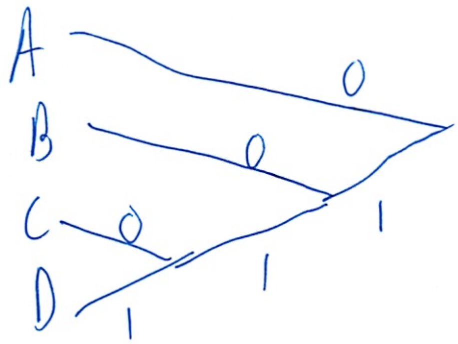
Take the 2 most least common probability



~~Done!~~

Then assign 0 and 1 to each
- can do it randomly
- but generally 0 going up

③



So

A 0
B 10
C 110
D 111

$\frac{1}{2}$ the time saved 1 bit

$\frac{1}{3}$ the time gained 1 bit

So $\frac{1}{6} \cdot 1$ bit - so net savings

To find out if good:

Compute avg lenght of code

$$\frac{1}{2}(1) + \frac{1}{6}(2) + \frac{1}{3}(3) = 2 - \frac{1}{6} = \frac{11}{6}$$

The log in formula is because adding bits ↑ possibilities exponentially

So best theoritical $= \frac{7}{4}$ ← not achievable nessarily

Fixed method $= 2$ ← if all probabilities are equal

Hellman $= \frac{11}{6}$ ← better method for this probability

(4)

Tutorial problems

1. Fish

50% bass

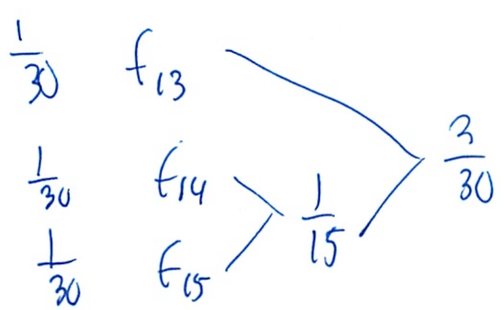
$1/30$ f_1

f_2

$\vdots$

f_{15}

How many bits to Huffman encode whole thing?
- diff question than online



1 2 3

2 fishes 1 fish 1 fish

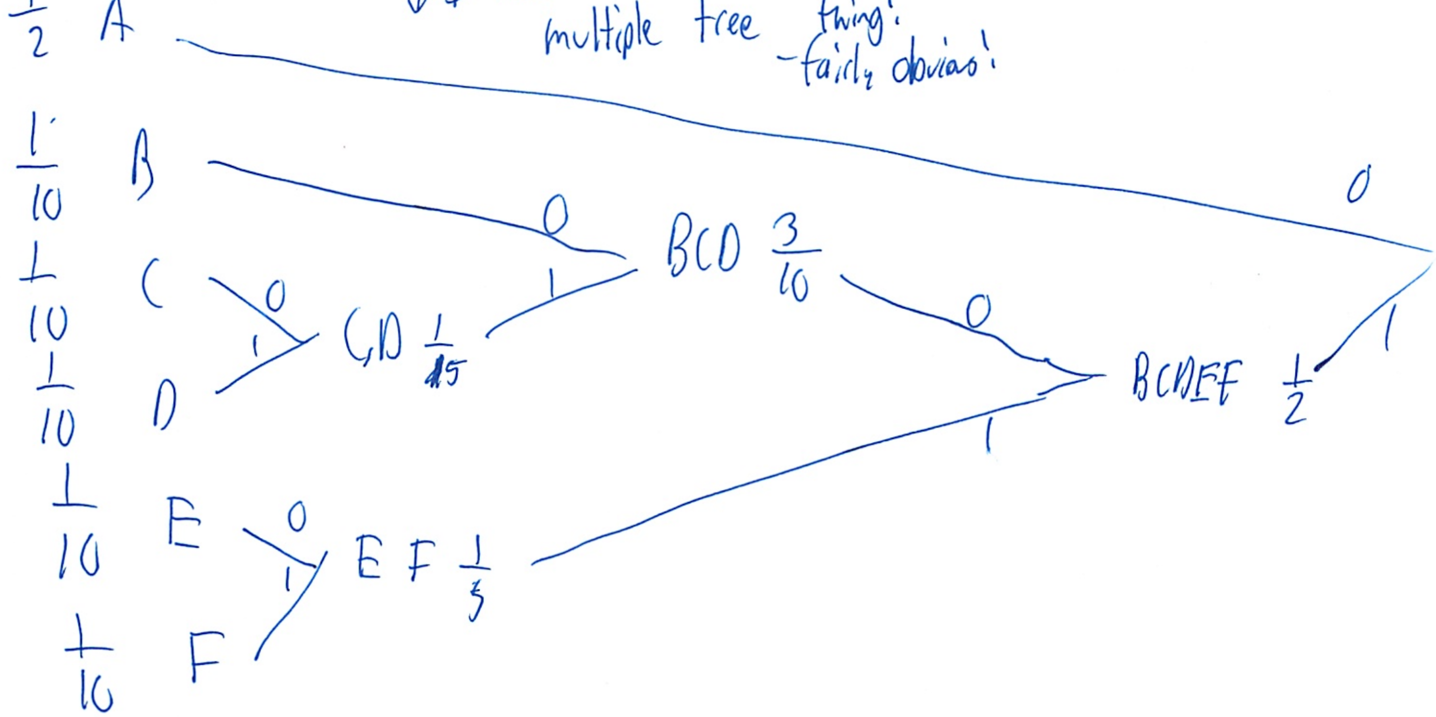
16 fish -1 = # steps (15)

(this is fun)

(5)

Oh wait TA doing differently
- 5 fishes

↓ I did not ~~[illegible]~~ realize can do multiple tree thing!
- fairly obvious!



$\frac{1}{2} \cdot 1$ bit
$\frac{1}{5}$ 3 bit
$\frac{1}{10}$ 3 bit
$\frac{1}{5}$ 4 bit

find weighted avg

(This is like Tabosh IAG 1 8th Grade problems)

Roughly try for $p \approx \frac{1}{2^k}$ - ~~[illegible]~~ will not use more than that bits

Unclear exactly how many bits - When prob are that - Huffman will achieve entropy

(6)

6. Online cold game

100 cards

- 3 aces
- 7 kings
- 25 queens
- 31 J
- 34 tens

Reshuffled each round

Bet on which card is drawn

So simple probability for each type

How much info ~~did~~ do you recieve when told Q drawn

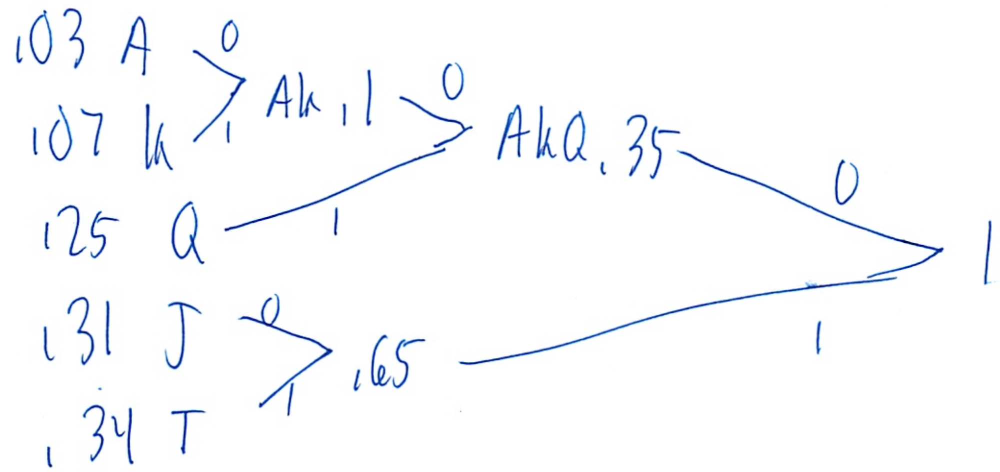
a) Theoritical info needed that got queen $\log \frac{1}{25/100} = \log_2 4 = 2$ bits

b) Compute entropy for entire thing

$.03 \cdot \log \frac{1}{.03} + .07 \log \frac{1}{.07} + \ldots\ldots = 1.97$ bits

c) Make a Huffman Code

7



A	000
K	001
Q	01
J	10
T	11

Can calc message length

$.03 \cdot 3 + .07 \cdot 3 + .25 \cdot 2 + .31 \cdot 2 + .34 \cdot 2 =$

See how it compares to theoretical

d) Decode $\underbrace{01}_{Q}\,\underbrace{001}_{K}\,\underbrace{000}_{A}\,\underbrace{11}_{T}\,\underbrace{10}_{J}\,\underbrace{01}_{Q}$

- note only works forward →
- prefix free property
  - critical to be able to decode

e) Opps already did

$3 \cdot .1 + 2 \cdot .9 = 2.1$ avg length

So for 1000 symbols, transmit $1000 \cdot 2.1 = 2100$ bits

(8)

f) Nevada gaming commission can compress into 43 fits via LZW algorthm

- ~~entropy is 344 b~~
- entropy is lower bound to compression algortm
- is 1.97, so 197 bits

Do know prob densities
- is rigged

Can also get an extreamly unlikly atcome

MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2010 (Last update: November 29, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

Read 2/3/11

CHAPTER 1
Encoding Information

In this lecture and the next, we'll be looking into *compression* techniques, which attempt to encode a message so as to transmit the same information using fewer bits. We'll be studying *lossless compression* where the recipient of the message can recover the original message exactly.

There are several reasons for using compression:

- Shorter messages take less time to transmit and so the complete message arrives more quickly at the recipient. This is good for both the sender and recipient since it frees up their network capacity for other purposes and reduces their network charges. For high-volume senders of data (such as Google, say), the impact of sending half as many bytes is economically significant.

- Using network resources sparingly is good for *all* the users who must share the internal resources (packet queues and links) of the network. Fewer resources per message means more messages can be accommodated within the network's resource constraints.

- Over error-prone links with non-negligible bit error rates, compressing messages before they are channel-coded using error-correcting codes can help improve throughput because all the redundancy in the message can be designed in to improve error resilience, after removing any other redundancies in the original message. It is better to design in redundancy with the explicit goal of correcting bit errors, rather than rely on whatever sub-optimal redundancies happen to exist in the original message.

Compression is traditionally thought of as an *end-to-end function*, applied as part of the application-layer protocol. For instance, one might use lossless compression between a web server and browser to reduce the number of bits sent when transferring a collection of web pages. As another example, one might use a compressed image format such as JPEG to transmit images, or a format like MPEG to transmit video. However, one may also apply compression at the link layer to reduce the number of transmitted bits and eliminate redundant bits (before possibly applying an error-correcting code over the link). When applied at the link layer, compression only makes sense if the data is inherently compressible, which means it cannot already be compressed and must have enough redundancy to extract compression gains.

1.1 Fixed-length vs. Variable-length Codes

Many forms of information have an obvious encoding, e.g., an ASCII text file consists of sequence of individual characters, each of which is independently encoded as a separate byte. There are other such encodings: images as a raster of color pixels (e.g., 8 bits each of red, green and blue intensity), sounds as a sequence of samples of the time-domain audio waveform, etc. What makes these encodings so popular is that they are produced and consumed by our computer's peripherals – characters typed on the keyboard, pixels received from a digital camera or sent to a display, digitized sound samples output to the computer's audio chip.

All these encodings involve a sequence of fixed-length symbols, each of which can be easily manipulated independently: to find the 42^{nd} character in the file, one just looks at the 42^{nd} byte and interprets those 8 bits as an ASCII character. A text file containing 1000 characters takes 8000 bits to store. If the text file were HTML to be sent over the network in response to an HTTP request, it would be natural to send the 1000 bytes (8000 bits) exactly as they appear in the file.

But let's think about how we might compress the file and send fewer than 8000 bits. If the file contained English text, we'd expect that the letter e would occur more frequently than, say, the letter x. This observation suggests that if we encoded e for transmission using *fewer* than 8 bits—and, as a trade-off, had to encode less common characters, like x, using more than 8 bits—we'd expect the encoded message to be shorter *on average* than the original method. So, for example, we might choose the bit sequence 00 to represent e and the code 100111100 to represent x. The mapping of information we wish to transmit or store to bit sequences to represent that information is referred to as a *code*. When the mapping is performed at the source of the data, generally for the purpose of *compressing* the data, the resulting mapping is called a *source code*. Source codes are distinct from *channel codes* we studied in Chapters 6–10: source codes *remove redundancy* and compress the data, while channel codes *add redundancy* to improve the error resilience of the data.

and have to encode when letter over

We can generalize this insight about encoding common symbols (such as the letter e) more succinctly than uncommon symbols into a strategy for *variable-length codes*:

> Send commonly occurring symbols using shorter codes (fewer bits) and infrequently occurring symbols using longer codes (more bits).

We'd expect that, on the average, encoding the message with a variable-length code would take fewer bits than the original fixed-length encoding. Of course, if the message were all x's the variable-length encoding would be longer, but our encoding scheme is designed to optimize the expected case, not the worst case.

Here's a simple example: suppose we had to design a system to send messages containing 1000 6.02 grades of A, B, C and D (MIT students rarely, if ever, get an F in 6.02 ◡̈). Examining past messages, we find that each of the four grades occurs with the probabilities shown in Figure 1-1.

Grade	Probability	Fixed-length Code	Variable-length Code
A	1/3	00	10
B	1/2	01	0
C	1/12	10	110
D	1/12	11	111

Figure 1-1: Possible grades shown with probabilities, fixed- and variable-length encodings

With four possible choices for each grade, if we use the fixed-length encoding, we need 2 bits to encode a grade, for a total transmission length of 2000 bits when sending 1000 grades.

Fixed-length encoding for $BCBAAB$: 01 10 01 00 00 01 (12 bits)

With a fixed-length code, the size of the transmission doesn't depend on the actual message – sending 1000 grades always takes exactly 2000 bits.

Decoding a message sent with the fixed-length code is straightforward: take each pair of message bits and look them up in the table above to determine the corresponding grade. Note that it's possible to determine, say, the 42^{nd} grade without decoding any other of the grades – just look at the 42^{nd} pair of bits.

Using the variable-length code, the number of bits needed for transmitting 1000 grades depends on the grades.

Variable-length encoding for $BCBAAB$: 0 110 0 10 10 0 (10 bits)

If the grades were all B, the transmission would take only 1000 bits; if they were all C's and D's, the transmission would take 3000 bits. But we can use the grade probabilities given in Figure 1-1 to compute the expected length of a transmission as

$$1000[(\frac{1}{3})(2) + (\frac{1}{2})(1) + (\frac{1}{12})(3) + (\frac{1}{12})(3)] = 1000[1\frac{2}{3}] = 1666.7 \text{ bits}$$

but still above theoretical $p \log_2(\frac{1}{p})$

So, on the average, using the variable-length code would shorten the transmission of 1000 grades by 333 bits, a savings of about 17%. Note that to determine, say, the 42^{nd} grade we would need to first decode the first 41 grades to determine where in the encoded message the 42^{nd} grade appears.

Using variable-length codes looks like a good approach if we want to send fewer bits but preserve all the information in the original message. On the downside, we give up the ability to access an arbitrary message symbol without first decoding the message up to that point.

One obvious question to ask about a particular variable-length code: is it the best encoding possible? Might there be a different variable-length code that could do a better job, i.e., produce even shorter messages on the average? How short can the messages be on the average?

1.2 How Much Compression Is Possible?

Ideally we'd like to design our compression algorithm to produce as few bits as possible: just enough bits to represent the information in the message, but no more. How do we measure the *information content* of a message? Claude Shannon proposed that we define information as a mathematical quantity expressing the probability of occurrence of a particular sequence of symbols as contrasted with that of alternative sequences.

Suppose that we're faced with N equally probable choices and we receive information that narrows it down to M choices. Shannon offered the following formula for the information received:

$$\log_2(N/M) \text{ bits of information} \tag{1.1}$$

Information is measured in *bits*, which you can interpret as the number of binary digits required to encode the choice(s). Some examples:

one flip of a fair coin
Before the flip, there are two equally probable choices: heads or tails. After the flip, we've narrowed it down to one choice. Amount of information = $\log_2(2/1) = 1$ bit.

roll of two dice
Each die has six faces, so in the roll of two dice there are 36 possible combinations for the outcome. Amount of information = $\log_2(36/1) = 5.2$ bits.

learning that a randomly-chosen decimal digit is even
There are ten decimal digits; five of them are even (0, 2, 4, 6, 8). Amount of information = $\log_2(10/5) = 1$ bit.

learning that a randomly-chosen decimal digit ≥ 5
Five of the ten decimal digits are greater than or equal to 5. Amount of information = $\log_2(10/5) = 1$ bit.

learning that a randomly-chosen decimal digit is a multiple of 3
Four of the ten decimal digits are multiples of 3 (0, 3, 6, 9). Amount of information = $\log_2(10/4) = 1.322$ bits.

need to study/process

learning that a randomly-chosen decimal digit is even, ≥ 5 and a multiple of 3
Only one of the decimal digits, 6, meets all three criteria. Amount of information = $\log_2(10/1) = 3.322$ bits. Note that this is same as the sum of the previous three examples: information is cumulative if there's no redundancy.

We can generalize equation (1.1) to deal with circumstances when the N choices are not equally probable. Let p_i be the probability that the i^{th} choice occurs. Then the amount of information received when learning of choice i is

$$\text{Information from } i^{th} \text{ choice} = \log_2(1/p_i) \text{ bits} \tag{1.2}$$

More information is received when learning of an unlikely choice (small p_i) than learning of a likely choice (large p_i). This jibes with our intuition about compression developed in §1.1: commonly occurring symbols have a higher p_i and thus convey less information, so we'll use fewer bits when encoding such symbols. Similarly, infrequently occurring symbols have a lower p_i and thus convey more information, so we'll use more bits when encoding such symbols. This exactly matches our goal of matching the size of the transmitted data to the information content of the message.

We can use equation (1.2) to compute the information content when learning of a choice by computing the weighted average of the information received for each particular choice:

$$\text{Information content in a choice} = \sum_{i=1}^{N} p_i \log_2(1/p_i) \tag{1.3}$$

This quantity is referred to as the *information entropy* or *Shannon's entropy* and is a lower bound on the amount of information which must be sent, on the average, when transmitting data about a particular choice.

What happens if we violate this lower bound, i.e., we send fewer bits on the average than called for by equation (1.3)? In this case the receiver will not have sufficient information and there will be some remaining ambiguity – exactly what ambiguity depends on the encoding, but in order to construct a code of fewer than the required number of bits, some of the choices must have been mapped into the same encoding. Thus, when the recipient receives one of the overloaded encodings, it doesn't have enough information to tell which of the choices actually occurred.

Equation (1.3) answers our question about how much compression is possible by giving us a lower bound on the number of bits that must be sent to resolve all ambiguities at the recipient. Reprising the example from Figure 1-1, we can update the figure using equation (1.2):

Grade	p_i	$log_2(1/p_i)$
A	1/3	1.58 bits
B	1/2	1 bit
C	1/12	3.58 bits
D	1/12	3.58 bits

Figure 1-2: Possible grades shown with probabilities and information content

Using equation (1.3) we can compute the information content when learning of a particular grade:

weighted avg

$$\sum_{i=1}^{N} p_i \log_2(\frac{1}{p_i}) = (\frac{1}{3})(1.58) + (\frac{1}{2})(1) + (\frac{1}{12})(3.58) + (\frac{1}{12})(3.58) = 1.626 \text{ bits}$$

So encoding a sequence of 1000 grades requires transmitting 1626 bits on the average. The variable-length code given in Figure 1-1 encodes 1000 grades using 1667 bits on the average, and so doesn't achieve the maximum possible compression. It turns out the example code does as well as possible when encoding one grade at a time. To get closer to the lower bound, we would need to encode sequences of grades – more on this below.

Finding a "good" code – one where the length of the encoded message matches the information content – is challenging and one often has to think outside the box. For example, consider transmitting the results of 1000 flips of an unfair coin where probability of heads is given by p_H. The information content in an unfair coin flip can be computed using equation (1.3):

$$p_H \log_2(1/p_H) + (1 - p_H) \log_2(1/(1 - p_H))$$

For $p_H = 0.999$, this evaluates to .0114. Can you think of a way to encode 1000 unfair coin flips using, on the average, just 11.4 bits? The recipient of the encoded message must be able to tell for each of the 1000 flips which were heads and which were tails. Hint: with a budget of just 11 bits, one obviously can't encode each flip separately!

One final observation: effective codes leverage the context in which the encoded message is being sent. For example, if the recipient is expecting to receive a Shakespeare sonnet, then it's possible to encode the message using just 8 bits if one knows that there are only 154 Shakespeare sonnets.

1.3 Huffman Codes

Let's turn our attention to developing an efficient encoding given a list of symbols to be transmitted and their probabilities of occurrence in the messages to be encoded. We'll use what we've learned above: more likely symbols should have short encodings, less likely symbols should have longer encodings.

If we diagram the variable-length code of Figure 1-1 as a binary tree, we'll get some insight into how the encoding algorithm should work:

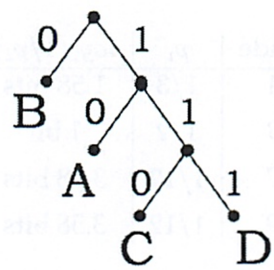


Figure 1-3: Variable-length code from Figure 1-1 diagrammed as binary tree

To encode a symbol using the tree, start at the root (the topmost node) and traverse the tree until you reach the symbol to be encoded – the encoding is the concatenation of the branch labels in the order the branches were visited. So B is encoded as 0, C is encoded as 110, and so on. Decoding reverses the process: use the bits from encoded message to guide a traversal of the tree starting at the root, consuming one bit each time a branch decision is required; when a symbol is reached at a leaf of the tree, that's next decoded message symbol. This process is repeated until all the encoded message bits have been consumed. So 111100 is decoded as: $111 \to D$, $10 \to A$, $0 \to B$.

Looking at the tree, we see that the most-probable symbols (e.g., B) are near the root of the tree and so have short encodings, while less-probable symbols (e.g., C or D) are further down and so have longer encodings. David Huffman used this observation to devise an algorithm for building the decoding tree for an *optimal* variable-length code while writing

(I like the notes)

a term paper for a graduate course here at M.I.T. The codes are optimal in the sense that there are no other variable-length codes that produce, on the average, shorter encoded messages. Note there are many equivalent optimal codes: the 0/1 labels on any pair of branches can be reversed, giving a different encoding that has the same expected length.

Huffman's insight was the build the decoding tree *bottom up* starting with the least-probable symbols. Here are the steps involved, along with a worked example based on the variable-length code in Figure 1-1:

programmers' spec

1. Create a set S of tuples, each tuple consists of a message symbol and its associated probability.

   Example: $S \leftarrow \{(0.333, A), (0.5, B), (0.083, C), (0.083, D)\}$

2. Remove from S the two tuples with the smallest probabilities, resolving ties arbitrarily. Combine the two symbols from the tuples to form a new tuple (representing an interior node of the decoding tree) and compute its associated probability by summing the two probabilities from the tuples. Add this new tuple to S.

   Example: $S \leftarrow \{(0.333, A), (0.5, B), (0.167, C \wedge D)\}$ $\cap$?

3. Repeat step 2 until S contains only a single tuple representing the root of the decoding tree.

   Example, iteration 2: $S \leftarrow \{(0.5, B), (0.5, A \wedge (C \wedge D))\}$
   Example, iteration 3: $S \leftarrow \{(1.0, B \wedge (A \wedge (C \wedge D)))\}$

Voila! The result is the binary tree representing an optimal variable-length code for the given symbols and probabilities. As you'll see in the Exercises the trees aren't always "tall and thin" with the left branch leading to a leaf; it's quite common for the trees to be much "bushier." – in recitation 7 today

With Huffman's algorithm in hand, we can explore more complicated variable-length codes where we consider encoding pairs of symbols, triples of symbols, quads of symbols, etc. Here's a tabulation of the results using the grades example:

Size of grouping	Number of leaves in tree	Expected length for 1000 grades
1	4	1667
2	16	1646
3	64	1637
4	256	1633

← can do better

Figure 1-4: Results from encoding more than one grade at a time

even though grades random

We see that we can approach the Shannon lower bound of 1626 bits for 1000 grades by encoding grades in larger groups at a time, but at a cost of a more complex encoding and decoding process.

We conclude with some observations about Huffman codes:

- Given static symbol probabilities, the Huffman algorithm creates an optimal encoding when each symbol is encoded separately. We can group symbols into larger meta-symbols and encode those instead, usually with some gain in compression but at a cost of increased encoding and decoding complexity.

- Huffman codes have the biggest impact on the average length of the encoded message when some symbols are substantially more probable than other symbols.
- Using *a priori* symbol probabilities (e.g., the frequency of letters in English when encoding English text) is convenient, but, in practice, symbol probabilities change message-to-message, or even within a single message.

The last observation suggests it would be nice to create an *adaptive* variable-length encoding that takes into account the actual content of the message. This is the subject of the next lecture.

but need to send along key/dict?

Exercises

Solutions to these exercises can be found in the tutorial problems for this lecture.

1. Huffman coding is used to compactly encode the species of fish tagged by a game warden. If 50% of the fish are bass and the rest are evenly divided among 15 other species, how many bits would be used to encode the species when a bass is tagged?

2. Several people at a party are trying to guess a 3-bit binary number. Alice is told that the number is odd; Bob is told that it is not a multiple of 3 (i.e., not 0, 3, or 6); Charlie is told that the number contains exactly two 1's; and Deb is given all three of these clues. How much information (in bits) did each player get about the number?

3. X is an unknown 8-bit binary number. You are given another 8-bit binary number, Y, and told that the Hamming distance between X (the unknown number) and Y (the number you know) is one. How many bits of information about X have you been given?

4. In Blackjack the dealer starts by dealing 2 cards each to himself and his opponent: one face down, one face up. After you look at your face-down card, you know a total of three cards. Assuming this was the first hand played from a new deck, how many bits of information do you have about the dealer's face down card after having seen three cards? $\log_2\left(\frac{52}{49}\right)$ ← possible remaining cards $52-3=49$

5. The following table shows the undergraduate and MEng enrollments for the School of Engineering.

Course (Department)	# of students	% of total
I (Civil & Env.)	121	7%
II (Mech. Eng.)	389	23%
III (Mat. Sci.)	127	7%
VI (EECS)	645	38%
X (Chem. Eng.)	237	13%
XVI (Aero & Astro)	198	12%
Total	1717	100%

(a) When you learn a randomly chosen engineering student's department you get some number of bits of information. For which student department do you get the least amount of information?

(b) Design a variable length Huffman code that minimizes the average number of bits in messages encoding the departments of randomly chosen groups of students. Show your Huffman tree and give the code for each course.

(c) If your code is used to send messages containing only the encodings of the departments for each student in groups of 100 randomly chosen students, what's the average length of such messages?

6. You're playing an on-line card game that uses a deck of 100 cards containing 3 Aces, 7 Kings, 25 Queens, 31 Jacks and 34 Tens. In each round of the game the cards are shuffled, you make a bet about what type of card will be drawn, then a single card is drawn and the winners are paid off. The drawn card is reinserted into the deck before the next round begins.

   (a) How much information do you receive when told that a Queen has been drawn during the current round?

   (b) Give a numeric expression for the information content received when learning about the outcome of a round.

   (c) Construct a variable-length Huffman encoding that minimizes the length of messages that report the outcome of a sequence of rounds. The outcome of a single round is encoded as A (ace), K (king), Q (queen), J (jack) or X (ten). Specify your encoding for each of A, K, Q, J and X.

   (d) Using your code from part (c) what is the expected length of a message reporting the outcome of 1000 rounds (i.e., a message that contains 1000 symbols)?

   (e) The Nevada Gaming Commission regularly receives messages in which the outcome for each round is encoded using the symbols $A, K, Q, J,$ and X. They discover that a large number of messages describing the outcome of 1000 rounds (i.e., messages with 1000 symbols) can be compressed by the LZW algorithm into files each containing 43 bytes in total. They decide to issue an indictment for running a crooked game. Why did the Commission issue the indictment?

7. Consider messages made up entirely of vowels (A, E, I, O, U). Here's a table of probabilities for each of the vowels:

l	p_l	$\log_2(1/p_l)$	$p_l \log_2(1/p_l)$
A	0.22	2.18	0.48
E	0.34	1.55	0.53
I	0.17	2.57	0.43
O	0.19	2.40	0.46
U	0.08	3.64	0.29
Totals	1.00	12.34	2.19

   (a) Give an expression for the number of bits of information you receive when learning that a particular vowel is either I or U.

(b) Using Huffman's algorithm, construct a variable-length code assuming that each vowel is encoded individually. Please draw a diagram of the Huffman tree and give the encoding for each of the vowels.

(c) Using your code from part (B) above, give an expression for the expected length in bits of an encoded message transmitting 100 vowels.

(d) Ben Bitdiddle spends all night working on a more complicated encoding algorithm and sends you email claiming that using his code the expected length in bits of an encoded message transmitting 100 vowels is 197 bits. Would you pay good money for his implementation?

Log Review

· The power to which the base must be raised to produce that #

(makes a lot more sense now!)

So $\log_2 16$ is ~~x~~ $2^x = 16$

$x = 4$? ✓

(Why did it take me 4+ years to figure that out!)

Properties

$$\log_b(xy) = \log_b(x) + \log_b(y)$$

$$\log_b(x^p) = p \log_b x$$ ← need to remember stuff

base 10 = common log

base e = natural log

↳ inverse function

$$e^{\ln(x)} = x$$

$$\ln(e^x) = x$$

(how do you find manually? w/ graph?)

(2)

More identities

$$\log_b\left(\frac{x}{y}\right) = \log_b(x) - \log_b(y)$$

$$\log_b\left(\sqrt[p]{x}\right) = \frac{\log_b(x)}{p}$$

Base change

$$\log_b(x) = \frac{\log_k x}{\log_k b}$$ where k is arbitrary

2/3

To save your work, click the SAVE button at the bottom of this page. You can revisit this page, revise your answers and SAVE as often as you like.

To submit the assignment, click the SUBMIT button at the bottom of this page. YOU CAN SUBMIT ONLY ONCE. Once the assignment has been submitted, you can continue to view this page but will no longer be able to make any changes to your answers.

6.02 Spring 2011: Plasmeier,Michael E.

PSet PS1

Dates & Deadlines

issued: Jan-29-2011 at 00:00
due: Feb-10-2011 at 06:00 (Feb-15-2011 at 06:00 with extension)
checkoff due: Feb-15-2011 at 06:00

Help is available from the staff in the 6.02 lab (38-530) during lab hours -- for the staffing schedule please see the Lab Hours page on the course website. You can also try an email to 6.02-help@mit.edu, although it's hard to debug code by trading emails!

Problem 1. Quickies (3 points)

2/3

A. I randomly select a letter from the 26-letter alphabet and tell you that my letter is *not* X, Y, or Z. How much information have I told you about my letter? Give the number of bits to 3 decimal places. Note that you can use Python to calculate the log base 2 of an argument x: `math.log(x,2)`.

23 chara alpha
log(23,2) = $\frac{\log(23)}{\log(2)}$ = 4.524

remaining

amt reduced ~~log(26)~~ ~~log(~~

log(26) − log(23)

= $\log\left(\frac{26}{23}\right)$

(points: 1)

B. You are trying to guess a card picked at random from a standard 52-card deck. Sam tells you the card is a spade; ~~Nora tells you it's not an ace;~~ Rita tells you its a seven. What is the total amount of information about the card given by Sam, Nora, and Rita? Give the number of bits to 3 decimal places.

7 of spades

exact card from 52

1 from 52 ~ log(52,2) 5.700

(points: 1)

C. You need to send a message listing the flip-by-flip results of 1024 independent flips of an unfair coin with p(heads) = 0.9 and p(tails) = 0.1. Based on the entropy of this distribution, what is a lower bound on the number of bits the message must contain? Please round up to the nearest integer. Note that there may not necessarily be an encoding that achieves this lower bound.

in notes

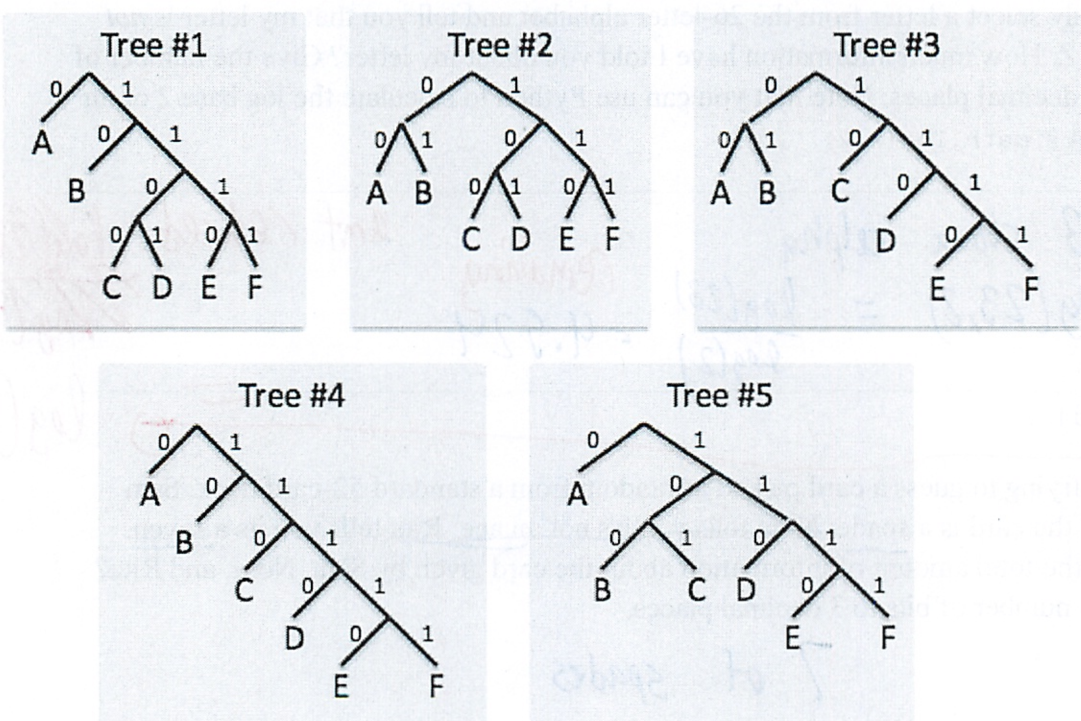
$.9 \log_2(\frac{1}{.9}) + .1 \log_2(\frac{1}{.1})$

$.468 \cdot 1024$

(points: 1) 480.252 if not do graping

Problem 2. (2 points)

Consider the five 6-leaf binary trees shown below, each of which diagrams a particular Huffman code for message sequences composed from six symbols A, B, C, D, E, F . Each symbol has an associated probability p(A), p(B), p(C), p(D), p(E), p(F).



A. Which tree or trees are consistent with a Huffman code where p(A) > 0.5?

first leaf
1, 4, 5

(points: 1)

B. Which tree or trees are consistent with a Huffman code where all six of the probabilities are equal?

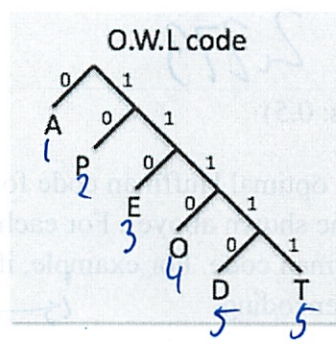
random
but try to spread out
2 is best

(points: 1)

Problem 3. (3 points)

The Hogwarts Registrar encodes the results of the O.W.L.s using the variable-length code shown below, next to the table of showing the probability that a student will receive a particular grade.

Grade	p(Grade)
O – outstanding	0.10
E – exceeds expectations	0.15
A – acceptable	0.40
P – poor	0.21
D – dreadful	0.09
T – troll	0.05



A. Decode the following OWL-encoded message from the Registrar:

11111110010111101111 1

T EAP O T

(points: 0.5)

B. What is the number of bits of information received when learning that a particular grade is passing (i.e., one of O, E, or A)? Please give your answer to three decimal

if even $\frac{3}{1/6} = \frac{6}{3}$ ← possibilities
← dn have info on

places.

$$\log_2\left(\frac{1}{.1+.15+.4}\right) = .621$$

(points: 0.5)

C. The Registrar is encoding a message containing 1000 O.W.L. grades.

1. What is the length in bits of the shortest and longest encoded messages that might be produced? not avg

~~Shortest~~ longest all trolls ✓ $5 \cdot 1000 =$

shortest all A $1 \cdot 1000 =$

(points: 0.5)

if not ~~binaries~~ string wise

2. What is the expected length of the encoded message?

Switch back + forth

- know what asking for!

avg

$.1 \log_2\left(\frac{1}{.1}\right) + .15 \log_2\left(\frac{1}{.15}\right) + .4$ ...etc

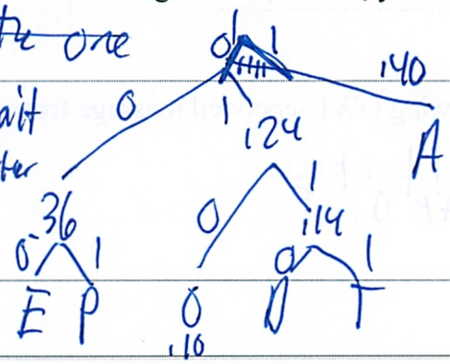
2.273

(points: 0.5)

D. Consider the optimal Huffman code for encoding O.W.L. grades (which may or may not be the one shown above). For each grade, give the length of its encoding in the optimal Huffman code. For example, if the encoding for P was "10", you'd give 2 as the length of its encoding.

is ~~the one~~

~~So just lengths from chart~~ wait better



A 1
E 3
P 3
O 3
D 4
T 4

(points: 1)

Python Task 1: Creating Huffman codes (8 points)

Useful download links:

PS1_tests.py -- test jigs for this assignment

Can check if better

$.4 \cdot 1 + (.15+.1+.21) \cdot 3 + (.09+.05) \cdot 4 =$

$= 2.34$

vs their one

2.37 not much better!

PS1_1.py -- template file for this task

2/6

The process of creating a variable-length code starts with a list of message symbols and their probabilities of occurrence. As described in the lecture notes, our goal is to encode more probable symbols with shorter binary sequences, and less probable symbols with longer binary sequences. The Huffman algorithm builds the binary tree representing the variable-length code from the bottom up, starting with the least probable symbols.

I was thinking of building one

Please complete the implementation of a Python function to build a Huffman code from a list of probabilities and symbols:

`(encoding_dictionary,tree) = huffman(plist)` 2

Given `plist`, a sequence of tuples `(prob,symbol)`, use the Huffman algorithm to build the binary tree representing an optimal variable-length code for messages consisting of the listed symbols. Use instances of the `Tree` class to represent leaves and interior nodes of the tree.

After the tree has been constructed, perform a recursive walk of the tree to build an encoding dictionary that maps symbols to their corresponding Huffman code.

Return a tuple containing the encoding dictionary and a `Tree` instance representing the root of the binary tree.

The template includes code for the `Tree` class and a start at the `huffman` function. You should complete the definition of the function by repeatedly processing the list of `Tree` instances, `tlist`, until `tlist` contains only a single instance -- the root of the Huffman tree. On each pass, remove the two `Tree` instances that have the smallest probability, construct a new `Tree` instance representing an interior node of the tree with the two instances as its children, computing the appropriate probability for the interior node, and add this new instance back into `tlist`.

Celearn lang

Specs in reading

The Python module `heapq` implements a priority queue data structure that is particularly efficient at letting you repeatedly select the minimum element of the list. A heap queue is a list whose elements are organized so that removing the minimimum element is fast, taking constant time independent of the size of the list. Adding a new element takes an amount of time that is logarithmic in the size of the list.

Need to see what variables are

`heapq.heapify(list)` can be called to reorganize the elements of `list` so that they form a heap queue. Just the order of the elements is changed, the list is still a list after the call to `heapify`.

`heapq.heappop(list)` removes the minimum element from `list` and returns it.

`heapq.heappush(list,item)` adds `item` to the `list`. ; and resorts

The heap queue operations use the "<" operator to compare list elements, so we've defined how "<" works on `Tree` instances by adding a `__lt__` method to the `Tree` class.

lengthy (tough to read in all their code)

2/3 ; do now or do math diagnostic 1st

PS1_1.py is the template file for this problem:

```
# template file for PS1, Python Task 1
import heapq
import PS1_tests

# an object representing a node in a Huffman tree.
class Tree:
    def __init__(self,p,left,right=None):
        self.p = p           # probability associated with node
        self.left = left     # left child (any Python value if leaf)
        self.right = right   # right child (None if leaf)
        # depth ensures the algorithm prefers to combine shallow
        # trees when selected items of equal probability   <- good, for special
        self.depth = 1 if right is None \                   cases
                   else 1 + max(self.left.depth,self.right.depth)

    # compare two tree nodes, sorting first by probability then
    # by depth of tree.  This is the low-level function called
    # by min or the less-than operator when the arguments are
    # instances of Tree.
    def __lt__(self,other):
        return self.p < other.p or \ <- next line
               (self.p == other.p and self.depth < other.depth)

    # return True if this instance is a leaf of the tree
    def isLeaf(self):
        return self.depth == 1

    # recursive procedure to construct encoding dictionary
    # by walking the tree to find all the leaf nodes.
    def walk(self,encode_dict,prefix):
        if self.isLeaf():
            encode_dict[self.left] = prefix
        else:
            self.left.walk(encode_dict,prefix+[0])    } Simple and
            self.right.walk(encode_dict,prefix+[1])   } recursive

# arguments:
#   plist -- sequence of (probability,object) tuples
# return:
#   (dict,tree) where
#     dict is a dictionary mapping object -> binary encoding
#     tree is the Huffman tree built by the algorithm.
def huffman(plist):
    # initialize set of tree nodes as leaves of the tree
    tlist = [Tree(p,obj) for p,obj in plist]   <- each object is a tree

    # Build Huffman tree by processing tlist until there is only a
    # single tree object left in the list (ie, the root of the
    # Huffman tree).  Consider using the heapq module.  You can
    # make a new node in the Huffman tree by calling
    #     Tree(probability,left_child,right_child).
```

Came together fairly fast ~too fast!

```
    # ****** YOUR CODE HERE... ******

    # walk the Huffman tree, adding an entry to the encoding
    # dictionary each time we find a leaf
    root = tlist[0]
    encoding_dict = {}
    root.walk(encoding_dict,[])

    # return (encoding dictionary,huffman tree)
    return (encoding_dict,root)

if __name__ == '__main__':
    # test case 1: four symbols with equal probability
    PS1_tests.test_huffman(huffman,
                           # symbol probabilities
                           ((0.25,'A'),(0.25,'B'),(0.25,'C'),
                            (0.25,'D')),
                           # expected encoding lengths
                           ((2,'A'),(2,'B'),(2,'C'),(2,'D')))

    # test case 2: example from section 22.3 in notes
    PS1_tests.test_huffman(huffman,
                           # symbol probabilities
                           ((0.34,'A'),(0.5,'B'),(0.08,'C'),
                            (0.08,'D')),
                           # expected encoding lengths
                           ((2,'A'),(1,'B'),(3,'C'),(3,'D')))

    # test case 3: example from Exercise 5 in notes
    PS1_tests.test_huffman(huffman,
                           # symbol probabilities
                           ((0.07,'I'),(0.23,'II'),(0.07,'III'),
                            (0.38,'VI'),(0.13,'X'),(0.12,'XVI')),
                           # expected encoding lengths
                           ((4,'I'),(3,'II'),(4,'III'),
                            (1,'VI'),(3,'X'),(3,'XVI')))

    # test case 4: 3 flips of unfair coin
    phead = 0.9
    plist = []
    for flip1 in ('H','T'):
        p1 = phead if flip1 == 'H' else 1-phead
        for flip2 in ('H','T'):
            p2 = phead if flip2 == 'H' else 1-phead
            for flip3 in ('H','T'):
                p3 = phead if flip3 == 'H' else 1-phead
                plist.append((p1*p2*p3,flip1+flip2+flip3))
    expected_sizes = ((1,'HHH'),(3,'HTH'),(5,'TTT'))
    PS1_tests.test_huffman(huffman,plist,expected_sizes)
```

The testing code in the template runs your code through several test cases. You should see something like the following print-out (your encodings may be slightly different, although the length of the encoding for each of the symbols should match that shown below):

```
Huffman encoding:
   B = 00
   D = 01
   A = 10
   C = 11
  Expected length of encoding a choice = 2.00 bits
  Information content in a choice = 2.00 bits
Huffman encoding:
   A = 00
   D = 010
   C = 011
   B = 1
  Expected length of encoding a choice = 1.66 bits
  Information content in a choice = 1.61 bits
Huffman encoding:
   II = 000
   I = 0010
   III = 0011
   X = 010
   XVI = 011
   VI = 1
  Expected length of encoding a choice = 2.38 bits
  Information content in a choice = 2.30 bits
Huffman encoding:
   HHH = 0
   HHT = 100
   HTH = 101
   THH = 110
   HTT = 11100
   THT = 11101
   TTH = 11110
   TTT = 11111
  Expected length of encoding a choice = 1.60 bits
  Information content in a choice = 1.41 bits
```

When you're ready, please submit the file with your code using the field below.

File to upload for Task 1: Browse...

Python Task 2: Decoding Huffman-encoded messages (8 points)

Useful download links:

PS1_2.py -- template file for this task

Encoding a message is a one-liner using the encoding dictionary returned by the `huffman` routine -- just use the dictionary to map each symbol in the message to its binary encoding and then concatenate the individual encodings to get the encoded message:

– nice, simple code

```
def encode(encoding_dict,message):
    return numpy.concatenate([encoding_dict[obj]
                              for obj in message])
```

↓ flat dict

Decoding uses the Huffman tree, also returned by the `huffman` routine: use the bits from the encoded message to guide a traversal of the tree starting at the root, consuming one bit each time a branch decision is required. When the traversal reaches a leaf of the tree, that's the next decoded message symbol. This process is repeated until all the encoded message bits have been consumed.

Please write a Python function to decode an encoded message using the supplied Huffman tree:

decoded_message = `decode(`*huffman_tree,encoded_message*`)`

> `encoded_message` is a numpy array of binary values, as returned by the `encode` function shown above. `huffman_tree` is a `Tree` instance representing the root of the binary Huffman tree. For non-leaf nodes in the tree, the instance slots `left` and `right` access the two descendents of the node.
>
> The `isLeaf()` method can be called to determine if a `Tree` instance represents a leaf of the Huffman tree, in which case the `left` instance slot holds the value of the leaf symbol.
>
> Return the sequence of symbols representing the decoded message.

← here we want a Tree again

PS1_2.py is the template file for this problem:

```
# template file for PS1, Python Task 2
import numpy,random
import PS1_tests
from PS1_1 import huffman

# arguments:
#   encoded_message -- numpy array of 0's and 1's
#   huffman_tree -- instance of Tree, root of Huffman tree
# return:
#   sequence of decoded symbols
def decode(huffman_tree,encoded_message):
    result = []

    # Use successive bits from encoded_message to guide
    # traversal of huffman_tree until a leaf is reached.
    # The value of the left slot will be the next symbol
    # to be appended to result.  Repeat until all the
    # bits of encoded_message have been consumed.

    # your code here...
```

← uses previos answer

takes me a long time to finish

sub functions - recursive
check main one if leaf
how to do this??
keeps returning 0s - something is wrong

keep changing strategy
finally got it
- mix of procedural + recursive
- took me 1 hr (learning that thinking) (how to approach)

```
        # return the result sequence
        return result

if __name__ == '__main__':
    # start by building Huffman tree from probabilities
    plist = ((0.34,'A'),(0.5,'B'),(0.08,'C'),(0.08,'D'))
    cdict,tree = huffman(plist)

    # test case 1: decode a simple message
    message = ['A', 'B', 'C', 'D']
    encoded_message = PS1_tests.encode(cdict,message)
    decoded_message = decode(tree,encoded_message)
    assert message == decoded_message, \
           "Decoding failed: expected %s, got %s" % \
           (message,decoded_message)

    # test case 2: construct a random message and encode it
    message = [random.choice('ABCD') for i in xrange(100)]
    encoded_message = PS1_tests.encode(cdict,message)
    decoded_message = decode(tree,encoded_message)
    assert message == decoded_message, \
           "Decoding failed: expected %s, got %s" % \
           (message,decoded_message)

    print "Tests passed!"
```

When you're ready, please submit the file with your code using the field below.

File to upload for Task 2: Browse...

Python Task 3: Huffman codes in use: fax transmissions (6 points)

Useful download links:

PS1_3.py -- Python file for this task
PS1_fax_image.png -- fax image

A fax machine scans the page to be transmitted, producing row after row of pixels. Here's what our test text image looks like:

> Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam aliquet. Proin dapibus, lorem id interdum interdum, libero erat consequat risus, et vehicula eros lacus non nibh. Fusce suscipit, ipsum in porttitor tempor, odio purus tempor libero, vehicula feugiat nisl tellus eu ante. Maecenas euismod placerat lectus. Duis quis quam eu elit pellentesque varius. Etiam non pede a arcu euismod tempor. Etiam tincidunt egestas nunc. Fusce auctor semper tortor. Morbi dolor diam, condimentum id, volutpat a, sagittis a, sem. Praesent ac pede ac nisl aliquam varius. Vivamus lacinia, magna ut bibendum interdum, ligula eros posuere nisl, at eleifend sapien dui vel enim. Maecenas vitae pede. Praesent vestibulum elit.
>
> Maecenas justo nisi, ullamcorper id, congue ac, convallis eget, purus. Fusce vel augue ac velit faucibus fringilla. Nulla quis purus sed urna cursus euismod. Nullam in leo. Sed aliquet nisi sit amet lectus. Phasellus blandit accumsan libero. Morbi eros augue, laoreet ut, blandit non, malesuada quis, purus. Morbi et elit eget elit consectetuer pretium. Nullam gravida sem vel urna. Fusce lacinia venenatis felis. Quisque tortor lorem, porttitor non, consequat eu, consequat et, massa.
>
> Vestibulum nisl nisi, ultricies et, volutpat sit amet, tincidunt ac, diam. Nam vel dolor. Praesent ante neque, tincidunt eu, adipiscing eget, blandit ac, lacus. Nulla facilisi. In commodo semper mi. Aliquam erat volutpat. Aenean consectetuer arcu a arcu. Proin aliquet odio ut nunc. Phasellus vel sem. Nullam nec libero.

Instead of sending 1 bit per pixel, we can do a lot better if we think about transmitting the image in chunks, observing that in each chunk we have alternating runs of white and black pixels. What's your sense of the distribution of run lengths, for example when we arrange the pixels in one long linear array? Does it differ between white and black runs?

have not learned yet

Perhaps we can compress the image by using **run-length encoding**, where we send the lengths of the alternating white and black runs, instead of sending the pixel pattern directly. For example, consider the following representation of a 4x7 bit image (1=white, 0=black):

```
1 1 0 0 1 1 1
1 1 1 0 0 1 1
1 1 1 1 0 0 1
1 1 1 1 1 1 1
```

Oh alternates b/w

This bit image can be represented as a sequence of run lengths: [2,2,6,2,6,2,8]. If the receiver knows that runs alternate between white and black (with the first run being white) and that the width of the image is 7, it can easily reconstruct the original bit pattern.

It's not clear that it would take fewer bits to transmit the run lengths than to transmit the original image pixel-by-pixel -- that'll depend on how clever we are when we encode the lengths! If all run lengths are equally probable then a fixed-length encoding for the lengths (e.g., using 8 bits to transmit lengths between 0 and 255) is the best we can do. But if some run length values are more probable than others, we can use a variable-length Huffman code to send the sequence of run lengths using fewer bits than can be achieved with a fixed-length code.

PS1_3.py runs several encoding experiments, trying different approaches to using Huffman encoding to get the greatest amount of compression. As is often the case with developing a compression scheme, one needs to experiment in order to gain the necessary insights about the most compressible representation of the message (in this case the text image).

Please run PS1_3.py, look at the output it generates, and then tackle the questions below.

Here are the alternative encodings we'll explore:

need to install matplot lib
— lots of ways to do so

Baseline 0 -- Transmit the b/w pixels as individual bits

The raw image contains 250,000 black/white pixels (0 = black, 1 = white). We could obviously transmit the image using 250,000 bits, so this is the baseline against which we can measure the performance of all other encodings.

Baseline 1 -- Encode run lengths with fixed-length code

To explore run-length encoding, we've represented the image as a sequence of alternating white and black runs, with a maximum run size of 255. If a particular run is longer than 255, the conversion process outputs a run of length 255, followed by a run of length 0 of the opposite color, and then works on encoding the remainder of the run. Since each run length can be encoded in 8 bits, the total size of the fixed-length encoding is 8 times the number of runs.

Baseline 2 -- Lempel-Ziv compressed PNG file

The original image is stored in a PNG-format file. PNG offers lossless compression based on the Lempel-Ziv algorithm for adaptive variable-length encoding described in section 22.4. We'd expect this baseline to be very good since adaptive variable-length coding is one of the most widely-used compression techniques.

Experiment 1 -- Huffman-encoding runs

As a first compression experiment, try using encoding run lengths using a Huffman code based on the probability of each possible run length. The experiment prints the 10 most-probable run lengths and their probabilities.

Experiment 2 -- Huffman-encoding runs by color

In this experiment, we try using separate Huffman codes for white runs and black runs. The experiment prints the 10 most-probable run lengths of each color.

Experiment 3 -- Huffman-encoding run pairs

Compression is always improved if you can take advantage of patterns in the message. In our run-length encoded image, the simplest pattern is a white run of some length (the space between characters) followed by a short black run (the black pixels of one row of the character).

Experiment 4 -- Huffman-encoding 4x4 image blocks

In this experiment, the image is split into 4x4 pixel blocks and the sixteen pixels in each block are taken to be a 16-bit binary number (i.e., a number in the range 0x0000 to 0xFFFF). A Huffman code is used to encode the sequence of 16-bit values. This encoding considers the two-dimensional nature of the image, rather than thinking of all the pixels as a linear array.

The questions below will ask you analyze the results. In each of the experiments, look closely at the top 10 symbols and their probabilities. When you see a small number of symbols that account for most of the message (i.e., their probabilities are high), that's when you'd expect to get good compression from a Huffman code.

Here's the code for PS1_3.py:

```
# file for PS1, Python Task 3
import matplotlib.pyplot as p
import numpy,os
import PS1_tests
from PS1_1 import huffman
from PS1_2 import decode

if __name__ == '__main__':
    # read in the image, convert into vector of pixels
    img = p.imread('PS1_fax_image.png')
    nrows,ncols,pixels = PS1_tests.img2pixels(img)

    # convert the image into a sequence of alternating
    # white and black runs, with a maximum run length
    # of 255 (longer runs are converted into multiple
    # runs of 255 followed by a run of 0 of the other
    # color).  So each element of the list is a number
    # between 0 and 255.
    runs = PS1_tests.pixels2runs(pixels,maxrun=255)

    # now print out number of bits for pixel-by-pixel
    # encoding and fixed-length encoding for runs
    print "Baseline 0:"
    print "  bits to encode pixels:",pixels.size

    print "\nBaseline 1:"
    print "  total number of runs:",runs.size
    print "  bits to encode runs with fixed-length code:",\
          8*runs.size

    print "\nBaseline 2:"
    print "  bits in Lempel-Ziv compressed PNG file:",\
          os.stat('PS1_fax_image.png').st_size*8

    # Start by computing the probability of each run length
    # by simply counting how many of each run length we have
    plist = PS1_tests.histogram(runs)

    # Experiment 1: Huffman-encoding run lengths
```

```
    cdict,tree = huffman(plist)
    encoded_runs = numpy.concatenate([cdict[r] for r in runs])
    print "\nExperiment 1:"
    print "  bits when Huffman-encoding runs:",\
          len(encoded_runs)
    print "  Top 10 run lengths [probability]:"
    for i in xrange(10):
        print "    %d [%3.2f]" % (plist[i][1],plist[i][0])

    # Experiment 2: Huffman-encoding white runs, black runs
    plist_white = PS1_tests.histogram(runs[0::2])
    cwhite,tree_white = huffman(plist_white)
    plist_black = PS1_tests.histogram(runs[1::2])
    cblack,tree_black = huffman(plist_black)
    encoded_runs = numpy.concatenate(
        [cwhite[runs[i]] if (i & 1) == 0 else cblack[runs[i]]
         for i in xrange(len(runs))])
    print "\nExperiment 2:"
    print "  bits when Huffman-encoding runs by color:",\
          len(encoded_runs)
    print "  Top 10 white run lengths [probability]:"
    for i in xrange(10):
        print "    %d [%3.2f]" % (plist_white[i][1],
                                  plist_white[i][0])
    print "  Top 10 black run lengths [probability]:"
    for i in xrange(10):
        print "    %d [%3.2f]" % (plist_black[i][1],
                                  plist_black[i][0])

    # Experiment 3: Huffman-encoding run pairs
    # where each pair is (white run,black run)
    pairs = [(runs[i],runs[i+1]) for i in xrange(0,len(runs),2)]
    plist_pairs = PS1_tests.histogram(pairs)
    cpair,tree_pair = huffman(plist_pairs)
    encoded_pairs = numpy.concatenate([cpair[pair]
                                       for pair in pairs])
    print "\nExperiment 3:"
    print "  bits when Huffman-encoding run pairs:",\
          len(encoded_pairs)
    print "  Top 10 run-length pairs [probability]:"
    for i in xrange(10):
        print "    %s [%3.2f]" % (str(plist_pairs[i][1]),
                                  plist_pairs[i][0])

    # Experiment 4: Huffman-encoding 4x4 image blocks
    blocks = PS1_tests.pixels2blocks(pixels,nrows,ncols,4,4)
    plist_blocks = PS1_tests.histogram(blocks)
    cblock,tree_block = huffman(plist_blocks)
    encoded_blocks = numpy.concatenate([cblock[b] for b in blocks])
    print "\nExperiment 4:"
    print "  bits when Huffman-encoding 4x4 image blocks:",\
          len(encoded_blocks)
    print "  Top 10 4x4 blocks [probability]:"
    for i in xrange(10):
        print "    0x%04x [%3.2f]" % (plist_blocks[i][1],
```

```
                                          plist_blocks[i][0])

    """
    # make sure we didn't goof somehow => display decoded image
    decoded_blocks = decode(tree_block,encoded_blocks)
    decoded_pixels = PS1_tests.blocks2pixels(decoded_blocks,
                                             nrows,ncols,4,4)
    decoded_img = PS1_tests.pixels2img(decoded_pixels,nrows,ncols)
    p.figure()
    p.title('Image decoded from 4x4 encoded blocks')
    p.imshow(decoded_img)
    p.show()
    """
```

The questions below include the results of running PS1_3.py using a particular implementation of `huffman`. Your results should be similar.

A. `Baseline 1:`

```
  total number of runs: 37712
  bits to encode runs with fixed-length code: 301696
```

Since 301696 > 250000, using an 8-bit fixed-length code to encode the run lengths uses more bits than encoding the image pixel-by-pixel. What does this tell you about the distribution of run length values? Hint: When runs are longer than 8 bits, the fixed-length encoding would be shorter than the pixel-by-pixel encoding.

Image switches color frequently ~~more~~ avg string less than 8 bits
~~Some exceed 255 strings~~ ~~In diff places~~

(points: 1) Not distribution - since fixed that would still save

B. `Experiment 1:`

```
  bits when Huffman-encoding runs: 111656
  Top 10 run lengths [probability]:
    1 [0.39]
    2 [0.19]
    3 [0.13]
    4 [0.12]
    5 [0.05]
    7 [0.03]
    6 [0.02]
    8 [0.01]
    0 [0.01]
    255 [0.01]
```

How much compression did Huffman encoding achieve, expressed as the ratio of unencoded size to encoded size (aka the *compression ratio*)? Briefly explain why the Huffman code was able to achieve such good compression.

$\frac{111656}{301696} = 37\%$ ← wrong order $\frac{301696}{111656} = 2.7$

(points: 1)

learning how compression bits

Briefly explain why the probability of zero-length runs is roughly equal to the probability of runs of length 255.

the rule

(points: 1)

— but don't have to do this nessarilly

C.
```
Experiment 2:
  bits when Huffman-encoding runs by color: 95357
  Top 10 white run lengths [probability]:
    2 [0.25]
    4 [0.20]
    3 [0.19]
    5 [0.08]
    6 [0.05]
    7 [0.05]
    1 [0.04]
    8 [0.02]
    255 [0.02]
    10 [0.02]
  Top 10 black run lengths [probability]:
    1 [0.73]
    2 [0.13]
    3 [0.07]
    4 [0.03]
    0 [0.02]
    5 [0.01]
    7 [0.01]
    8 [0.00]
    9 [0.00]
    10 [0.00]
```

Briefly explain why the compression ratio is better in Experiment 2 than in Experiment 1.

White is often next to each other (negitive space)
while black is letters
Having ~~[illegible]~~ white text on black bg would

(points: 1)

make black the same as old white

D.
```
Experiment 3:
  bits when Huffman-encoding run pairs: 87310
  Top 10 run-length pairs [probability]:
    (2, 1) [0.20]
    (4, 1) [0.15]
    (3, 1) [0.12]
    (5, 1) [0.07]
    (3, 2) [0.04]
    (7, 1) [0.03]
    (2, 2) [0.03]
    (4, 2) [0.03]
    (1, 1) [0.03]
    (6, 1) [0.03]
```

Briefly explain why the compression ratio is better in Experiment 3 than in Experiments 1 and 2.

Same as instructions

(points: 1)

E.
```
Experiment 4:
  bits when Huffman-encoding 4x4 image blocks: 71628
  Top 10 4x4 blocks [probability]:
    0xffff [0.55]
    0xbbbb [0.02]
    0xdddd [0.02]
    0xeeee [0.01]
    0x7777 [0.01]
    0x7fff [0.01]
    0xefff [0.01]
    0xfff7 [0.01]
    0xfffe [0.01]
    0x6666 [0.01]
```

Using a Huffman code to encode 4x4 pixel blocks results in a better compression ratio than achieved even by PNG encoding. Briefly explain why. [Note that the number of bits reported for the Huffman-encoded 4x4 blocks does not include the cost of transmitting the custom Huffman code to the receiver, so the comparison is not really apples-to-apples. But ignore this for now -- one can still make a compelling argument as to why block-based encoding works better than sequential pixel encoding in the case of text images.]

Large sections of white space (55%!)

(points: 1)

You can save your work at any time by clicking the Save button below. You can revisit this page, revise your answers and SAVE as often as you like.

Save

To submit the assignment, click on the Submit button below. YOU CAN SUBMIT ONLY ONCE after which you will not be able to make any further changes to your answers. Once an assignment is submitted, solutions will be visible after the due date and the graders will have access to your answers. When the grading is complete, points and grader comments will be shown on this page.

Submit

✓ Submitted 2/6/2011
1:33pm

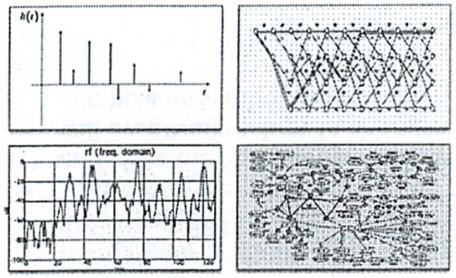
6.02 Checkoff 1

2/11

Post

Very easy
Went through my hw
Clarified 1 qu
Asked me a few more qu
that I was able to answer
Like making img smaller - ~~done~~
-w/ real text

Went nicely
Said P-Sets will be harder soon
~~A~~ Just like coding interviews
Was grading my P-set at same time



INTRODUCTION TO EECS II
DIGITAL COMMUNICATION SYSTEMS

6.02 Spring 2011 Lecture #2

- Adaptive variable-length codes: LZW
- Perceptual coding

- P set posted (I did)
- no fixed scheduled lab
- Lab 38-530

Checkoff interview is about code
Bring your code with you

Example from Last Lecture

entropy

$choice_i$	p_i	$log_2(1/p_i)$	p_i * $log_2(1/p_i)$	Huffman encoding	Expected length
"A"	1/3	1.58 bits	0.528 bits	10	0.667 bits
"B"	1/2	1 bit	0.5 bits	0	0.5 bits
"C"	1/12	3.58 bits	0.299 bits	110	0.25 bits
"D"	1/12	3.58 bits	0.299 bits	111	0.25 bits
			1.626 bits		1.667 bits

Entropy is 1.626 bits/symbol, expected length of Huffman encoding is 1.667 bits/symbol.

How do we do better?

16 Pairs: 1.646 bits/sym
64 Triples: 1.637 bits/sym
256 Quads: 1.633 bits/sym

└ Can't do too long!

1000 message
4^{1000} possibilities of orders

- Measuring info content
Could encode multiple symbols at a time
└ Huffman tree for pairs

Huffman Codes – the final word?

Used in tons of places

- Given static symbol probabilities, the Huffman algorithm creates an optimal encoding when each symbol is encoded separately. (optimal ≡ no other encoding will have a shorter expected message length)
- Huffman codes have the biggest impact on average message length when some symbols are substantially more likely than other symbols.
- You can improve the results by adding encodings for symbol pairs, triples, quads, etc. But the number of possible encodings quickly becomes intractable.
- Symbol probabilities change message-to-message, or even within a single message.
- Can we do adaptive variable-length encoding?

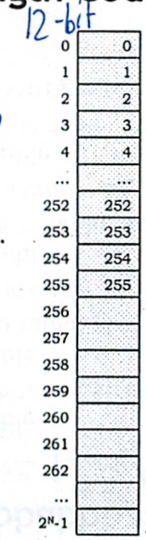
must send code along as well

and priorities might change

Can you build adaptive coder?

Adaptive Variable-length Codes

- Algorithm first developed by Lempel and Ziv, later improved by Welch. Now commonly referred to as the "LZW Algorithm" zip
- As message is processed a "string table" is built which maps symbol sequences to an N-bit fixed-length code. Table size = 2^N
- Transmit table indices, usually shorter than the corresponding string → compression!
- Note: String table can be reconstructed by the decoder based on information in the encoded stream – the table, while central to the encoding and decoding process, is never transmitted!



hopefully bottom part of table replaces long string

1/2

LZW Encoding

```
STRING = get input symbol
WHILE there are still input symbols DO
    SYMBOL = get input symbol
    IF STRING + SYMBOL is in the string table THEN
        STRING = STRING + SYMBOL
    ELSE
        output the code for STRING
        add STRING + SYMBOL to the string table
        STRING = SYMBOL
    END
END

output the code for STRING
```

at first seems dumb
but table gets populated quickly

1. Accumulate message bytes in S as long as S appears in table.
2. When S+b isn't in table: send code for S, add S+b to table.
3. Reinitialize S with b, back to step 1.

Example: Encode "abbbabbbab..."

256	ab
257	bb
258	bba
259	abb
260	bbab
261	
262	

1. Read a; string = a < 97 in ASCII
2. Read b; ab not in table
   output 97, add ab to table, string = b
3. Read b; bb not in table
   output 98, add bb to table, string = b
4. Read b; bb in table, string = bb
5. Read a; bba not in table
   output 257, add bba to table, string = a
6. Read b, ab in table, string = ab
7. Read b, abb not in table
   output 256, add abb to table, string = b
8. Read b, bb in table, string = bb
9. Read a, bba in table, string = bba
10. Read b, bbab not in table
    output 258, add bbab to table, string = b

+ entries get longer + longer

Encoder Notes

- The encoder algorithm is greedy – it's designed to find the longest possible match in the string table before it makes a transmission.
- The string table is filled with sequences actually found in the message stream. No encodings are wasted on sequences not actually found in the file.
- Note that in this example the amount of compression increases as the encoding progresses, i.e., more input bytes are consumed between transmissions.
- Eventually the table will fill and then be reinitialized, recycling the N-bit codes for new sequences. So the encoder will eventually adapt to changes in the probabilities of the symbols or symbol sequences.

remember: don't need to send table
Only knows of stuff already decoded

LZW Decoding

```
Read CODE
output CODE
STRING = CODE

WHILE there are still codes to receive DO
    Read CODE
    IF CODE is not in the translation table THEN
        ENTRY = STRING + STRING[0]
    ELSE
        ENTRY = get translation of CODE
    END
    output ENTRY
    add STRING+ENTRY[0] to the translation table
    STRING = ENTRY
END
```

Easy: use table lookup to convert code to message string
Less easy: build table that's identical to that in encoder

have to build table as well along with

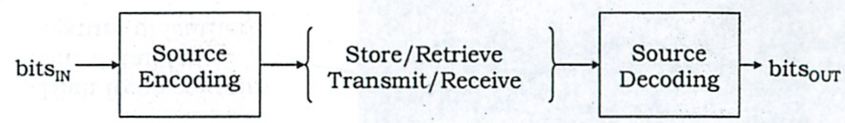
Example: Decode 97, 97, 257, 256, 258

256	ab
257	bb
258	bba
259	abb
260	
261	
262	

1. Read 97;
   output a; string = a
2. Read 98; entry = b
   output b; add ab to table; string = b
3. Read 257; entry = bb
   output bb; add bb to table; string = bb
4. Read 256; entry = ab
   output ab; add bba to table; string = ab
5. Read 258; entry = bba
   output bba; add abb to table; string = bba

...

Lossless vs. Lossy Compression



- Huffman and LZW encodings are *lossless*, i.e., we can reconstruct the original bit stream exactly: bits$_{OUT}$ = bits$_{IN}$.
  - What we want for "naturally digital" bit streams (documents, messages, datasets, ...)
- Any use for *lossy* encodings: bits$_{OUT}$ ≈ bits$_{IN}$?
  - "Essential" information preserved
  - Appropriate for sampled bit streams (audio, video) intended for human consumption via imperfect sensors (ears, eyes). take advantage of human's capabilities

Perceptual Coding

- Start by evaluating input response of bitstream consumer (eg, human ears or eyes), i.e., how consumer will perceive the input. how people perceive things
  - Frequency range, amplitude sensitivity, color response, ...
  - Masking effects
- Identify information that can be removed from bit stream without perceived effect, e.g.,
  - Sounds outside frequency range, or masked sounds
  - Visual detail below resolution limit (color, spatial detail)
  - Info beyond maximum allowed output bit rate do remove extra detail to fit constraint
- Encode remaining information efficiently
  - Use DCT-based transformations (real instead of complex)
  - Quantize DCT coefficients
  - Entropy code (eg, Huffman encoding) results

more sensitive to grayscale - luminosity than color
edges

Perceptual Coding Example: Images

- Characteristics of our visual system ⇒ opportunities to remove information from the bit stream
  - More sensitive to changes in luminance than color ⇒ spend more bits on luminance than color (encode separately)
  - More sensitive to large changes in intensity (edges) than small changes ⇒ quantize intensity values
  - Less sensitive to changes in intensity at higher spatial frequencies ⇒ use larger quanta at higher spatial frequencies
- So to perceptually encode image, we would need:
  - Intensity at different spatial frequencies
  - Luminance (grey scale intensity) separate from color intensity

If values close together
Send 1 value to correspond to all of them

JPEG Image Compression

JPEG = Joint Photographic Experts Group

8x8 blocks
- HDTV
just 1 color
until more detail
comes in

Lenna Söderberg, Miss November 1972

everyone uses this pic to test compression

RGB to YCbCr Conversion → Group into 8x8 blocks of pixels → Convert to energy at different spatial freqs. → Quantizer → Entropy Encoder → 011010...

Performed for each 8x8 block of pixels

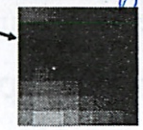
YCbCr Color Representation

JPEG-YCbCr (601) from "digital 8-bit RGB"

Y = 16 + 0.299*R + 0.587*G + 0.114*B
Cb = 128 - 0.168736*R - 0.331264*G + 0.5* B
Cr = 128 + 0.5*R - 0.418688*G - 0.081312*B

All values are in the range 16 to 235

luminosity
blue
green
4 times the bandwidth

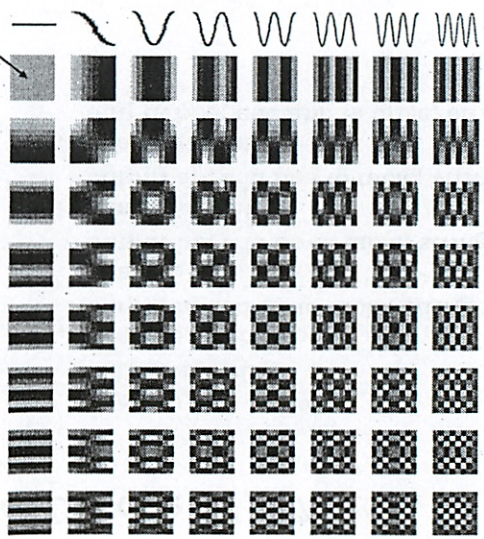


8x8 block of pixels

http://en.wikipedia.org/wiki/YCrCb

2D DCT Basis Functions

DC Component



Want to do better

all the basis functions

So find these basis blocks and pick a few and multiply by constants to get exactly an 8x8 block

high spacial freq.
lose fine detail — hair fuzzy, etc

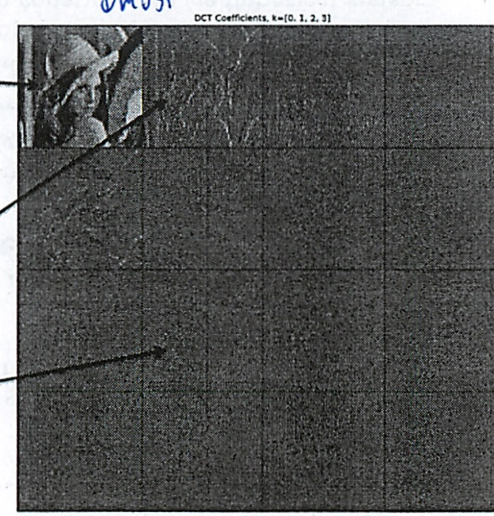
Lenna DCT Coeffs from each 8x8 block

almost just single 8x8 block

DC coeffs carry a lot of the picture info!

Low freqs contain major edge info (note more H and V...)

High freqs contain fine detail (eg, texture of feather)



hair showing up

very little info here

hard to see on printout

Quantization (the "lossy" part)

then Huffman encode

Divide each of the 64 DCT coefficients by the appropriate quantizer value (Q_{lum} for Y, Q_{chr} for Cb and Cr) and round to nearest integer ⇒ many 0 values, many of the rest are small integers.

$$Q_{lum} = \begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix}$$

Color

$$Q_{chr} = \begin{pmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{pmatrix}$$

throw away a lot of info

Note fewer quantization levels in Q_{chr} and at higher spatial frequencies. Change "quality" by choosing different quantization matrices.

scale - for better quality make # smaller

divide by #
the size of the buckets
throwing away fine detail

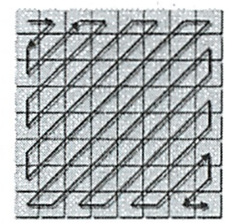
Quantization Example

```
[[-231 -148   38  -24  -15    0    4    0]
 [ 153  -11  -35   -2  -28   14   -2    0]
 [   3   73  -16  -29    2    8   -4   -3]
 [  -4   28   17  -25   -1    6   -8   -4]
 [   0    4    5    6    4    4   -2   -5]
 [   3   -4    2   10    6    0   -6   -3]
 [  -2    0   -1    6    3   -1   -5   -5]
 [  -3    1    2   -2    0    1    0    0]]
```

DCT Coefficients

```
[[-14 -13   4  -2  -1   0   0   0]
 [ 13  -1  -2   0  -1   0   0   0]
 [  0   6  -1  -1   0   0   0   0]
 [  0   2   1  -1   0   0   0   0]
 [  0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0]]
```

Quantized/Rounded Coefficients



Visit coeffs in order of increasing spatial frequency ⇒ tends to create long runs of 0s towards end of list:

```
-14
-13 13
  0 -1  4
 -2 -2  6  0
  0  2 -1  0 -1
  0 -1 -1  1  0  0
  0  0  0 -1  0  0  0...
```


Entropy Encoding Example

then huffman encode

Quantized coeffs:

`-14 -13 13 0 -1 4 -2 -2 6 0 0 2 -1 0 -1 0 -1 -1 1 0 0 0 0 0 -1 0 0 0...`

DC: (N),coeff, all the rest: (run,N),coeff

```
(4)-14 (0,4)-13 (0,4)13 (1,1)-1 (0,3)4 (0,2)-2 (0,2)-2
(0,3)6 (2,2)2 (0,1)-1 (1,1)-1 (0,1)-1 (0,1)1 (5,1)-1 EOB
```

Encode using Huffman codes for N and (run,N):

```
1010001 10110010 10111101 11000 100100 0101 0101
100110 1111101110 000 11000 000 001 11110100 1010
```

Result: 8x8 block of 8-bit pixels (512 bits) encoded as 84 bits

6x compression!

To read more see "The JPEG Still Picture Compression Standard" by Gregory K. Wallace
http://white.stanford.edu/~brian/psy221/reader/Wallace.JPEG.pdf

JPEG Results

the darker the dot the greater the error

The source image (left) was converted to JPEG (q=50) and then compared, pixel-by-pixel. The error is shown in the right-hand image (darker = larger error).

- fine detail

http://en.wikipedia.org/wiki/JPEG

MIT 6.02 **DRAFT** *Lecture Notes*
Spring 2011
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

CHAPTER 2
Compression

2.1 Adaptive Variable-length Codes

One approach to adaptive encoding is to use a two pass process: in the first pass, count how often each symbol (or pairs of symbols, or triples – whatever level of grouping you've chosen) appears and use those counts to develop a Huffman code customized to the contents of the file. Then, on a second pass, encode the file using the customized Huffman code. This is an expensive but workable strategy, yet it falls short in several ways. Whatever size symbol grouping is chosen, it won't do an optimal job on encoding recurring groups of some different size, either larger or smaller. And if the symbol probabilities change dramatically at some point in the file, a one-size-fits-all Huffman code won't be optimal; in this case one would want to change the encoding midstream.

A somewhat different approach to adaptation is taken by the popular Lempel-Ziv-Welch (LZW) algorithm. As the message to be encoded is processed, the LZW algorithm builds a *string table* which maps symbol sequences to/from an N-bit index. The string table has 2^N entries and the transmitted code can be used at the decoder as an index into the string table to retrieve the corresponding original symbol sequence. The sequences stored in the table can be arbitrarily long, so there's no *a priori* limit to the amount of compression that can be achieved. The algorithm is designed so that the string table can be reconstructed by the decoder based on information in the encoded stream – the table, while central to the encoding and decoding process, is never transmitted!

When encoding a byte stream, the first 256 entries of the string table are initialized to hold all the possible one-byte sequences. The other entries will be filled in as the message byte stream is processed. The encoding strategy works as follows (see the pseudo-code in Figure 3-1): accumulate message bytes as long as the accumulated sequence appears as some entry in the string table. At some point appending the next byte b to the accumulated sequence S would create a sequence $S + b$ that's not in the string table. The encoder then

- transmits the N-bit code for the sequence S.
- adds a new entry to the string table for $S + b$. If the encoder finds the table full when it goes to add an entry, it reinitializes the table before the addition is made.

```
initialize TABLE[0 to 255] = code for individual bytes
STRING = get input symbol
while there are still input symbols:
   SYMBOL = get input symbol
   if STRING + SYMBOL is in TABLE:  ; String + Symbol
       STRING = STRING + SYMBOL
   else:
       output the code for STRING
       add STRING + SYMBOL to TABLE
       STRING = SYMBOL
output the code for STRING
```

Figure 2-1: Pseudo-code for LZW adaptive variable-length encoder. Note that some details, like dealing with a full string table, are omitted for simplicity.

```
initialize TABLE[0 to 255] = code for individual bytes
CODE = read next code from encoder
STRING = TABLE[CODE]
output STRING

while there are still codes to receive:
   CODE = read next code from encoder        must rebuild
   if TABLE[CODE] is not defined:
       ENTRY = STRING + STRING[0]
   else:
       ENTRY = TABLE[CODE]
   output ENTRY
   add STRING+ENTRY[0] to TABLE
   STRING = ENTRY
```

Figure 2-2: Pseudo-code for LZW adaptive variable-length decoder

- resets S to contain only the byte b.

This process is repeated until all the message bytes have been consumed, at which point the encoder makes a final transmission of the N-bit code for the current sequence S.

Note that for every transmission a new entry is made in the string table. With a little cleverness, the decoder (see the pseudo-code in Figure 3-2) can figure out what the new entry must have been as it receives each N-bit code. With a duplicate string table at the decoder, it's easy to recover the original message: just use the received N-bit code as index into the string table to retrieve the original sequence of message bytes.

Figure 3-3 shows the encoder in action on a repeating sequence of *abc*. Some things to notice:

- The encoder algorithm is greedy – it's designed to find the longest possible match in the string table before it makes a transmission.

abcabcabcabc ...

S	msg. byte	lookup	result	transmit	string table
–	a	–	–	–	–
a	b	ab	not found	index of a	table[256] = ab
b	c	bc	not found	index of b	table[257] = bc
c	a	ca	not found	index of c	table[258] = ca
a	b	ab	found	–	–
ab	c	abc	not found	256	table[259] = abc
c	a	ca	found	–	–
ca	b	cab	not found	258	table[260] = cab
b	c	bc	found	–	–
bc	a	bca	not found	257	table[261] = bca
a	b	ab	found	–	–
ab	c	abc	found	–	–
abc	a	abca	not found	259	table[262] = abca
a	b	ab	found	–	–
ab	c	abc	found	–	–
abc	a	abca	found	–	–
abca	b	abcab	not found	262	table[263] = abcab
b	c	bc	found	–	–
bc	a	bca	found	–	–
bca	b	bcab	not found	261	table[264] = bcab
b	c	bc	found	–	–
bc	a	bca	found	–	–
bca	b	bcab	found	–	–
bcab	c	bcabc	not found	264	table[265] = bcabc
c	a	ca	found	–	–
ca	b	cab	found	–	–
cab	c	cabc	not found	260	table[266] = cabc
c	a	ca	found	–	–
ca	b	cab	found	–	–
cab	c	cabc	found	–	–
cabc	a	cabca	not found	266	table[267] = cabca
a	b	ab	found	–	–
ab	c	abc	found	–	–
abc	a	abca	found	–	–
abca	b	abcab	found	–	–
abcab	c	abcabc	not found	263	table[268] = abcabc
c	– end –	–	–	index of c	–

← last char + current

how is it greedy? Oh does not send anything – keeps looking till string no longer matches – sends matching part, adds new part to table puts first letter as string

Figure 2-3: LZW encoding of string "abcabcabcabcabcabcabcabcabcabcabcabc"

received	string table	decoding
a	–	a
b	table[256] = ab	b
c	table[257] = bc	c
256	table[258] = ca	ab
258	table[259] = abc	ca
257	table[260] = cab	bc
259	table[261] = bca	abc
262	table[262] = abca	abca
261	table[263] = abcab	bca
264	table[264] = bacb	bcab
260	table[265] = bcabc	cab
266	table[266] = cabc	cabc
263	table[267] = cabca	abcab
c	table[268] = abcabc	c

adds to table

? did I mess up past? since did too early?

Figure 2-4: LZW decoding of the sequence $a, b, c, 256, 258, 257, 259, 262, 261, 264, 260, 266, 263, c$

- The string table is filled with sequences actually found in the message stream. No encodings are wasted on sequences not actually found in the file.
- Since the encoder operates without any knowledge of what's to come in the message stream, there may be entries in the string table that don't correspond to a sequence that's repeated, i.e., some of the possible N-bit codes will never be transmitted. This means the encoding isn't optimal – a prescient encoder could do a better job.
- Note that in this example the amount of compression increases as the encoding progresses, i.e., more input bytes are consumed between transmissions.
- Eventually the table will fill and then be reinitialized, recycling the N-bit codes for new sequences. So the encoder will eventually adapt to changes in the probabilities of the symbols or symbol sequences.

Figure 3-4 shows the operation of the decoder on the transmit sequence produced in Figure 3-3. As each N-bit code is received, the decoder deduces the correct entry to make in the string table (i.e., the same entry as made at the encoder) and then uses the N-bit code as index into the table to retrieve the original message sequence.

Some final observations on LZW codes:

- a common choice for the size of the string table is 4096 ($N = 12$). A larger table means the encoder has a longer memory for sequences it has seen and increases the possibility of discovering repeated sequences across longer spans of message. This is a two-edged sword: dedicating string table entries to remembering sequences that will never been seen again decreases the efficiency of the encoding.
- Early in the encoding, we're using entries near the beginning of the string table, i.e., the high-order bits of the string table index will be 0 until the string table starts to fill. So the N-bit codes we transmit at the outset will be numerically small. Some variants of LZW transmit a variable-width code, where the width grows as the table

Huffman? - no it knows current table size fills. If $N = 12$, the initial transmissions may be only 9 bits until the 511^{th} entry of the table is filled, then the code exands to 10 bits, and so on until the maximum width N is reached.

- Some variants of LZW introduce additional special transmit codes, e.g., CLEAR to indicate when the table is reinitialized. This allows the encoder to reset the table preemptively if the message stream probabilities change dramatically causing an observable drop in compression efficiency.

- There are many small details we haven't discussed. For example, when sending N-bit codes one bit at a time over a serial communication channel, we have to specify the order in the which the N bits are sent: least significant bit first, or most significant bit first. To specify N, serialization order, algorithm version, etc., most compressed file formats have a header where the encoder can communicate these details to the decoder.

Exercises

1. Describe the contents of the string table created when encoding a very long string of all a's using the simple version of the LZW encoder shown in Figure 3-1. In this example, if the decoder has received E encoded symbols (i.e., string table indices) from the encoder, how many a's has it been able to decode?

2. Consider the pseudo-code for the LZW decoder given in Figure 3-1. Suppose that this decoder has received the following five codes from the LZW encoder (these are the first five codes from a longer compression run):

```
97 -- index of 'a' in the translation table
98 -- index of 'b' in the translation table
257 -- index of second addition to the translation table
256 -- index of first addition to the translation table
258 -- index of third addition to in the translation table
```

   After it has finished processing the fifth code, what are the entries in the translation table and what is the cumulative output of the decoder?

   table[256]: __________

   table[257]: __________

   table[258]: __________

   table[259]: __________

   cumulative output from decoder: ______________

6.02 Recitation

2/8

(20 min late) (half over)

LZW encoding

$k \cdot \log k$ bits used to encode

$a \to 1$
$aa \to 2$
$aaa \to 3$
$\vdots$
$\underbrace{aa\ldots a}_{k} \to k$

$\frac{k(k+1)}{2}$ for # of a's = k transmission

$\approx \frac{k \log_2 k \text{ bits transmitted}}{\left(\frac{k(k+1)}{2}\right)}$

$\approx \frac{\log_2 k}{k}$

(drop constants)

Information increases at $\log k$

° Aside (didn't really get - missed begining)

Output of a coding process should look random

- Or else you have some pattens you have not removed

Decoding

Can you construct table as you recieve bits?

String + next something was not in table

So add it to table

2

96 - - - a - - - - String

256 not in table

what to do now?

put a temp placeholder

String + n

String + n, n'

a n

? unknown

a n g so equals that

a n

know must equal string + string[0]

last transmission decoded

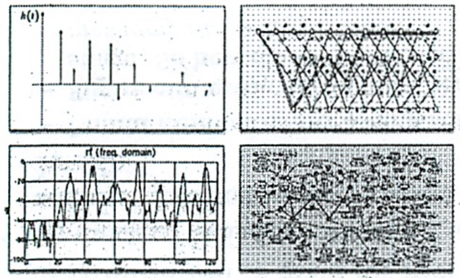
append first char of the string

Is code ∈ table

Yes → Output table[code] → assign to string

No → String + string[0]

(I don't think he explained it well - many people are confused)



INTRODUCTION TO EECS II

DIGITAL COMMUNICATION SYSTEMS

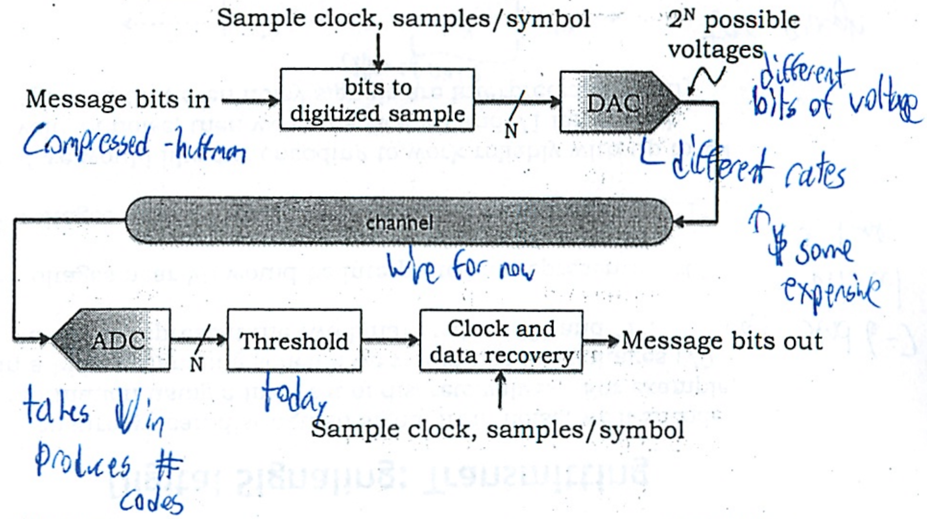
Start w/ perfect world
over next weeks
add noise/
real-world

6.02 Spring 2011
Lecture #3

- Analog woes, the digital abstraction
- Basic digital recipes for sending information

Real World

Diagram of a Communication Channel



Representing information with voltage

1920s and 30s TV

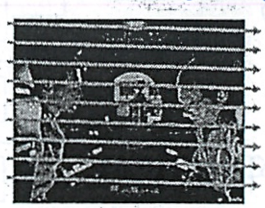
Representation of each point (x, y) on a B&W Picture:

intensity

0 volts: BLACK
1 volt: WHITE
0.37 volts: 37% Gray
etc.

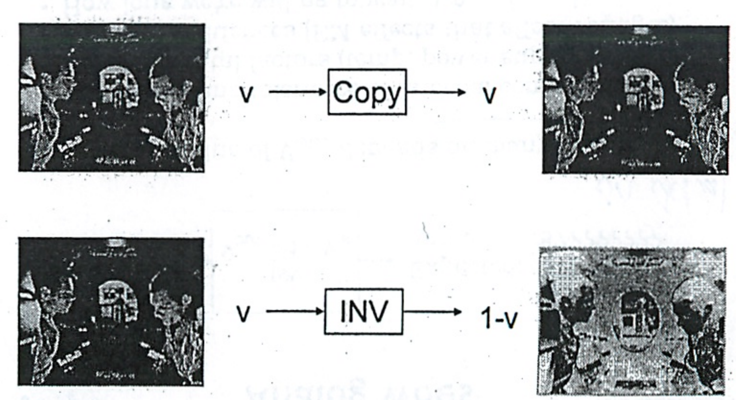
Continuous rows

Representation of a picture:
Scan points in some prescribed raster order... generate voltage waveform

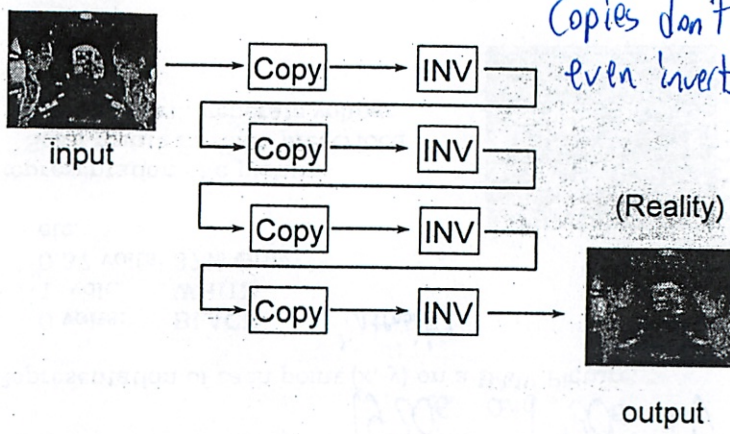


System Building Blocks

- First let's introduce some processing blocks:



Let's build a system!

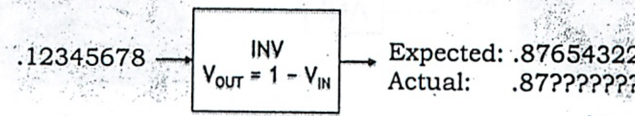


Copies don't do anything even inverts back to same

quality reduced not perfect

Why?

Analog Woes



The actual value of V_{OUT} depends on many factors: on actual comerical components

- Manufacturing tolerance of internal components
- Environmental factors (temp, power supply voltage)
- External influences (EM effects that affect voltages)
- How long we're willing to wait very long time - average, perfect
- How much we're willing to spend $

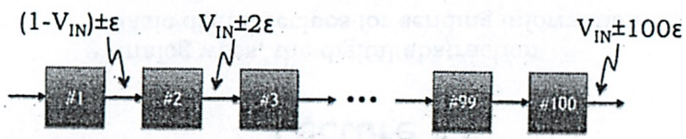
Truth in advertising: $V_{OUT} = (1 - V_{IN}) \pm \varepsilon$ If we call it ε maybe it'll seem small ☺

Can vary over a pretty big range

environmental conditions lots of EM sheilding

Analog Errors Accumulate

Now a permanate part of the signal



- If, say, ε = 1%, then result might be 100% off (urk!)
- Accumulation is good for money, bad for errors Compounding
- As system builders we want to guarantee output without having to worry about exact internal details don't want too many constraints
  - Bound number of processing stages in series OR
  - Figure out a way to eliminate errors at each processing stage. So how do we know *which part of the signal is message and which is error?*

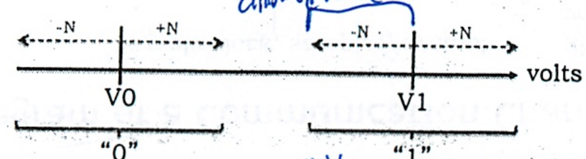
make things symple

Digital Signaling: Transmitting

To ensure we can distinguish signal from noise, we'll encode information using a fixed set of discrete values. For example, in a binary signaling scheme we would use two voltages (V0 and V1) to represent the two binary values "0" and "1". can send 6-7 bits at a time

- voltages near V0 would be interpreted as representing "0"
- voltages near V1 would be interpreted as representing "1"
- if we would like our encoding to work reliably with up to ±N volts of noise, then we can space V0 and V1 far enough apart so that even noisy signals are interpreted correctly

allowed noise



far enough apart

still distinguisable
↳ noise immunity - computers get much better at

(have seen this before ~~but~~ where?)

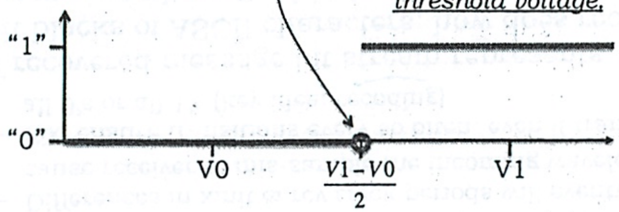
Digital Signaling: Receiving

We can specify the behavior of the receiver with a graph that shows how incoming voltages are mapped to "0" and "1".

Costs $ to do precision

One possibility:

The boundary between "0" and "1" regions is called the threshold voltage.



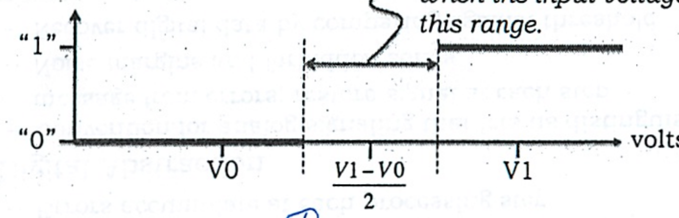
all possible V that could arrive

It would be hard to build a receiver that met this specification since it's very expensive and time consuming to accurately measure voltages (e.g., those near the threshold voltage).

We Need a "Forbidden Zone"

We need to change our specification to include a "forbidden zone" where there is *no mapping* between the continuous input voltage and the discrete output:

Receiver can output any value when the input voltage is in this range.

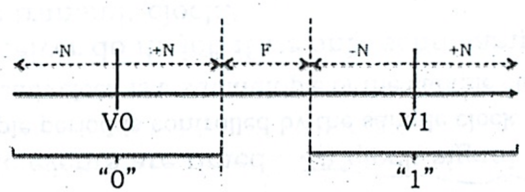


Now the specification has some "elbow room" which allows for manufacturing and environmental differences from receiver to receiver.

does not have to be precise

- the jump can be anywhere in the middle
- No spec what it has to do

Digital Signaling: Final Specification



Engineering tradeoffs when choosing F, the width in volts of the forbidden zone:

Smaller F: allows larger N (better noise tolerance), but receiver is more expensive to build (tighter manufacturing and environmental tolerances).

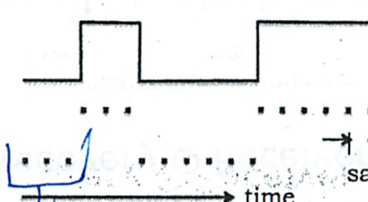
Larger F: less noise tolerance, but cheaper, faster receivers.

Digital Signaling in 6.02

- In 6.02 we'll represent voltage waveforms using sequences of voltage samples
  - Sample rate specifies the number of samples/second and hence the time interval between samples, e.g., 4e6 samples/second (4 Msps) means the time interval between samples is .25e-6 seconds (250ns).
  - Each transmission of a single bit ("0" or "1") will entail sending some number of consecutive voltage samples (V0 or V1 volts); we'll choose an appropriate number of samples/bit in each application. Goal: smaller is better!

Continuous time

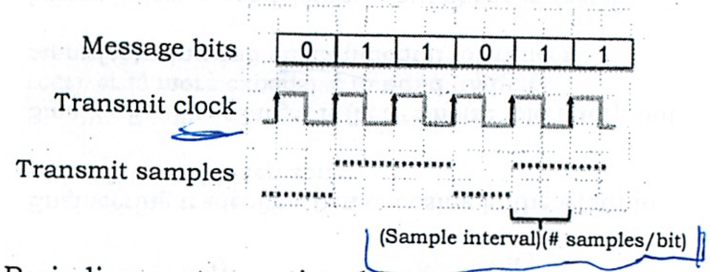
Discrete time



Samples

# of samples per bit – need length of sample

Transmitting Information

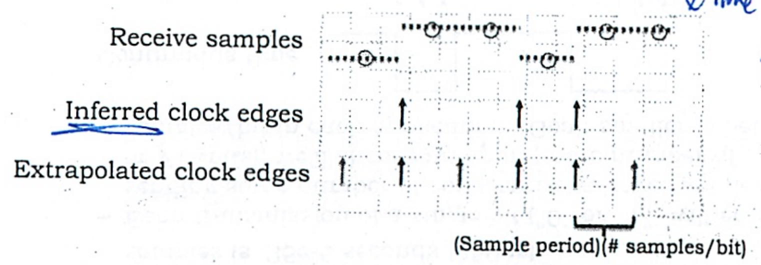


length of time per bit

- Periodic events are timed by a clock signal
  - Sample period is controlled by the sample clock
  - Transmit clock is a submultiple of the sample clock
- Can receiver do its job if we only send samples and not the transmit clock?
  - Save a wire and the power needed to drive clock signal

Clock Recovery @ Receiver

time base is slightly off



- Receiver can infer presence of clock edge every time there's a transition in the received samples.
- Using sample period, extrapolate remaining edges
  - Now know first and last sample for each bit
- Choose "middle" sample to determine message bit

Know samples per bit

"clock recovery"

more tomorrow - more robust way to recieve

Two Issues for Recitation

Next weeks lab

- Don't want receiver to extrapolate over too long an interval
  - Differences in xmit & rcv clock periods will eventually cause receiver to mis-sample the incoming waveform
  - Fix: ensure transitions every so often, even if transmitting all 0's or all 1's (key idea: recoding)
- If recovered message bit stream represents, say, 8-bit blocks of ASCII characters, how does receiver determine where the blocks start?
  - Need out-of-band information about block starts
  - Fix: use special bit sequences to periodically synchronize receiver's notion of block boundaries. These sync sequences must be unique (i.e., distinguishable from ordinary message traffic).

change how it is encoded

want to match 0 and 1s

So -5V 5V

No ~~current~~ transmited power

Need to know where groups are

L Could see intelligible English

2. Send a "sync" pattern — 8b/10b - 20% overhead

L must make sure not at junction by accident

Summary

- Analog signaling has issues
  - Real-world circuits & channels introduce errors
  - Errors accumulate at each processing step
- Digital Abstraction
  - Convention for analog signaling that lets us distinguish message from errors; restore signal at each step
  - Noise margins and forbidden zones
  - Recover digital data by comparing against threshold
- Receiver design
  - We don't send xmit clock, receiver does clock recovery
  - Determine bit from samples in "middle" of bit cell

New networks 64b/65b

2/10

6.02 Recitation

Transmitting Bits

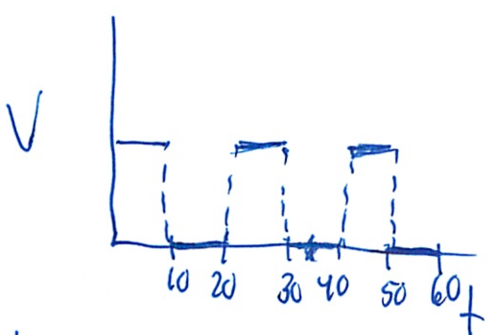
How will you send the bits?

- Wire - ~~has posit~~

But how do you do that?

- measure voltage on the wire
- but this is not exact

10 10 10



time lenght of bit is constant

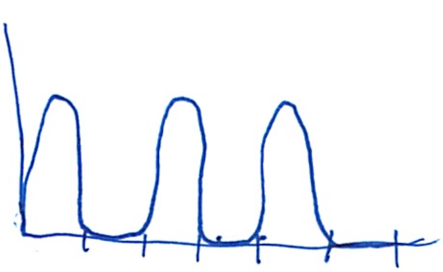
You can only sample discretely

- say every 2 seconds

||||| . . . . . |||||

↑ more error around transition

May be more like



②

So what do you do?

- take average, if > .5 its a 1
- just sample the middle one

Inference and estimation

<u>Practacilities</u>

Want to know when things start
And then go to the middle

Problems

1. How many samples per bit?

You sample at 2μs

Tx 1	Tx 2
10μs/bit	20μs/bit
5 samples/bit	10 samples/bit

2. Where does my byte start?
   - You just start listening randomly

2b How to ~~make sure~~ recover if your clock drifts

3 Solving

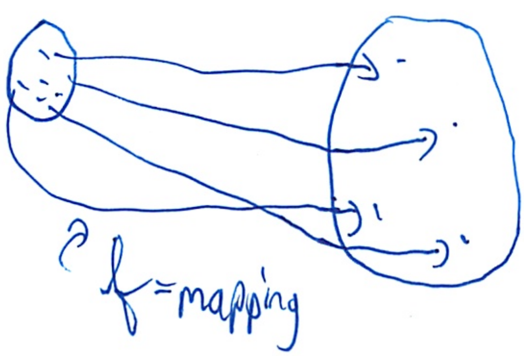
1. Can see where bit transmissions are within the samples
   - need lots of transitions to happen
2. Approximate balance of 0s and 1s
   - don't want to send current down the wire
   - "DC Balance"
3. Some special sync symbol

Solution

8 bit → 10 bit from IBM

Try coming up with your own

2^8 (256) ⟶ 2^{10} (1024)



Take any 8 bit sequence b, look at 10 bit map f(b)

If does not have

- more than 5 consecutive symbols
  - so freq transitions
- difference b/w 1s and 0s is ≤ 6 symbols in a certain lenght message

Create a mapping for each of the 256 bytes

4

It has a special sync symbol

0011111

1100000

Any concated combo of bytes won't make up this patter

example

$$\overbrace{11000011}^{A}\,\overbrace{11100101}^{B}$$ ← not allowed

$$1100\underbrace{0011\,111}_{\text{Sync Symbol "hidden" inside}}00101$$

So to find code

1. Find all the ones w/ equal # of 0s and 1s
   - get $\sim \frac{2^n}{\sqrt{n}} = \frac{2^{10}}{\sqrt{10}} \approx \frac{1}{3} \cdot 1024 \approx 340$ possible responses
2. Make sure sync is not show up
   - naive way $\binom{340}{256}$ combos
   - TA: think about a more efficent way

5

P-Set 2 Hints

#2

keep moving the search window →

$0\,1\,0\,1\,1\,0\,1\,\underbrace{0\,0\,1\,1\,1\,1}_{\text{Sync symbol}}\,\underbrace{0\,1\,0\,1}_{\text{message}}\rightarrow$

Use mapping

Send 16 messages

- Divide into 10 bit block
- Translate back into 8 bit blocks

Utilization

Was $16 \cdot 8$ bits $= 128$ bit

Us $20 + 7 + 160 = 187$ bits

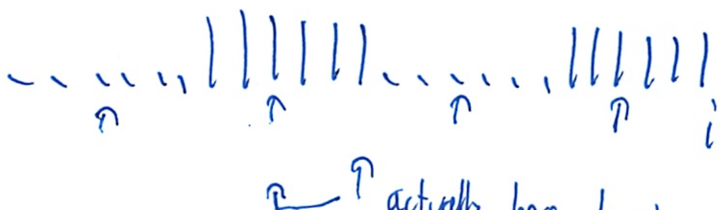
$$\frac{128}{137} \approx \frac{13}{19} \approx 70 \text{ bits}$$

6

#1. Physical world – sampling in the middle

How to know you are shifting away?

- look in the middle of 2 of your samples
- if diffreent from First, but = to 2nd
  - then you shifted



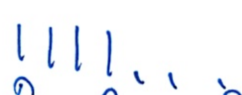
ideal world

actually here lagging

3-
is the same

- need a transition

Only if you see a transmition



look in middle

~~its a 2~~

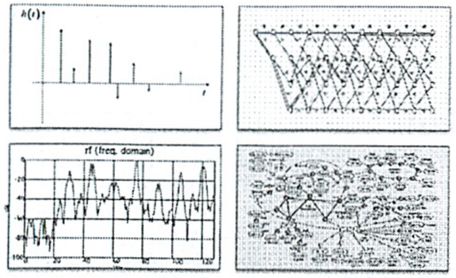
~~also~~ saw – make sure its a transition

1 its a 0 1
So now you know you are ~~slow~~ fast

if it was a 0 (the middle)

~~there~~ ~~if~~ know you know you are slow

keep adjusting between



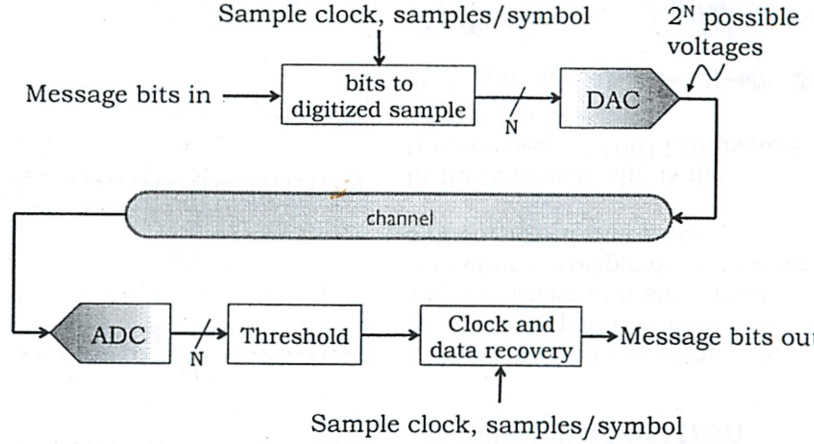
INTRODUCTION TO EECS II

DIGITAL COMMUNICATION SYSTEMS

6.02 Spring 2011
Lecture #4

- Inputs & responses
- Linear time-invariant systems
- Modeling communications channels

Today: Modeling Channel Behavior



Can predict what will come out
 └ build models
then improve models w/ insights

Linear Time-Invariant Systems

System Input and Response

input transmission channel response

$$x[n] \longrightarrow \boxed{S} \longrightarrow y[n]$$

X[17] = 17th symbol input

A discrete-time signal is described by an infinite sequence of values, denoted by x[n], y[n], z[n], and so on. The indices range from −∞ to +∞.

In the diagram above, the sequence of output values y[n] is called the *response* of system S to the *input* sequence x[n].

Unit Step and Unit Step Response

Special inputs - help us learn about system

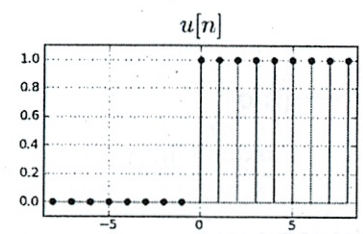
A simple but useful discrete-time signal is the *unit step*, u[n], defined as

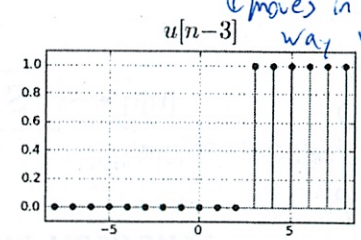
$$u[n] = \begin{cases} 0, & n < 0 \\ 1, & n \geq 1 \end{cases}$$

Unit step Unit step response

$$u[n] \longrightarrow \boxed{S} \longrightarrow s[n]$$

(moves in opposite way you think





any transmission can be made of steps + shifts

2/14

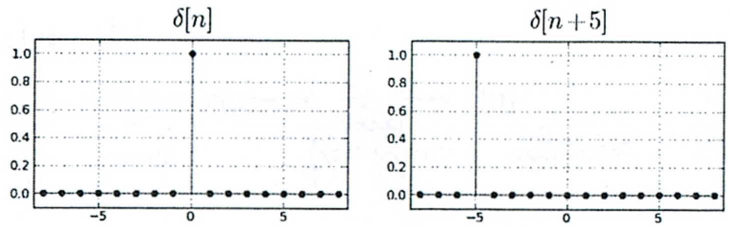
Unit Sample

even more basic

Another simple but useful discrete-time signal is the *unit sample*, δ[n], defined as

can build it as

$$\delta[n] = \underbrace{u[n]-u[n-1]}_{\text{can build it as}} = \begin{cases} 0, & n \neq 0 \\ 1, & n = 0 \end{cases}$$



Unit Sample Response

very special

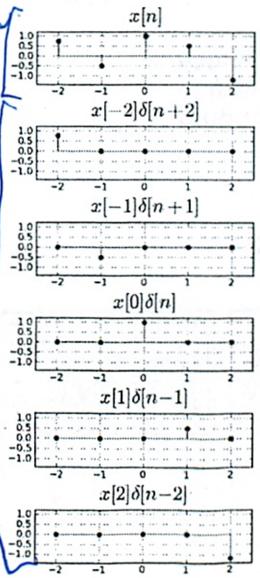
$$\underset{\text{Unit sample}}{\delta[n]} \longrightarrow \boxed{S} \longrightarrow \underset{\text{Unit sample response}}{h[n]}$$

Can tell you everything about it

The *unit sample response* of a system S is the response of the system to the unit sample input. We will always denote the unit sample response as h[n].

Unit-sample Decomposition

This seq is the sum of these seqs



A discrete-time signal can be decomposed into a sum of time-shifted, scaled unit samples.

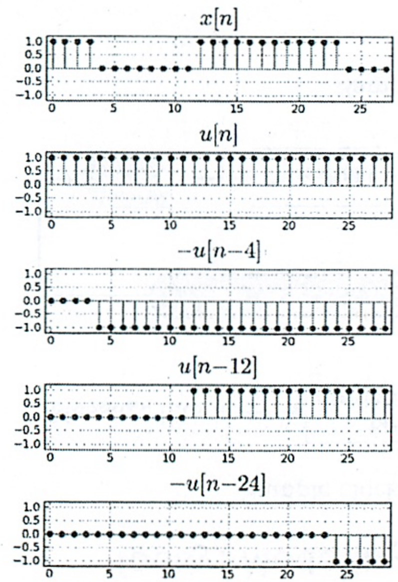
Example: in the figure, x[n] is the sum of x[-2]δ[n+2] + x[-1]δ[n+1] + … + x[2]δ[n-2].

In general:

scale, position

$$x[n] = \sum_{k=-\infty}^{\infty} x[k]\delta[n-k]$$

Unit-step Decomposition



Digital signaling waveforms are easily decomposed into time-shifted, scaled unit steps (each transition corresponds to another shifted, scaled unit step).

In this example, x[n] is the transmission of 1001110 using 4 samples/bit:

$$x[n] = u[n] - u[n-4] + u[n-12] - u[n-24]$$

represent any digital transmission as sum of scaled + shifted steps

Time Invariant Systems

Let y[n] be the response of S to input x[n].

If for all possible sequences x[n] and integers N

$$x[n-N] \longrightarrow \boxed{S} \longrightarrow y[n-N]$$

Can shift right or left indicies - no same

real world is kinda like this

then system S is said to be *time invariant.* A time shift in the input sequence to S results in an identical time shift of the output sequence.

Linear Systems

Let $y_1[n]$ be the response of S to input $x_1[n]$ and $y_2[n]$ be the response to $x_2[n]$.

If

any linear combo

$$ax_1[n]+bx_2[n] \longrightarrow \boxed{S} \longrightarrow ay_1[n]+by_2[n]$$

then system S is said to be *linear*. If the input is the weighted sum of several signals, the response is the *superposition* (i.e., weighted sum) of the response to those signals.

- real worlds happens like this

- does not have to be over every possible values
- only the ones you want to use
- not 1 trillion Volts

Modeling LTI Systems

If system S is both linear and time-invariant (LTI), then we can use the unit sample response to predict the response to any input waveform x[n]:

Sum of shifted, scaled unit samples Sum of shifted, scaled responses

$$x[n]=\sum_{k=-\infty}^{\infty} x[k]\delta[n-k] \longrightarrow \boxed{S} \longrightarrow y[n]=\sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

Convolution sum

Indeed, the unit sample response h[n] completely characterizes the LTI system S, so you often see

$$x[n] \longrightarrow \boxed{h_S[n]} \longrightarrow y[n]$$

Properties of Convolution

$$x[n]*h[n] \equiv \sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

The summation is called the convolution sum, or more simply, the *convolution* of x[n] and h[n]. "*" is the convolution operator.

Lstrugle for the symbols

Convolution is commutative:

$$x[n]*h[n]=h[n]*x[n]$$

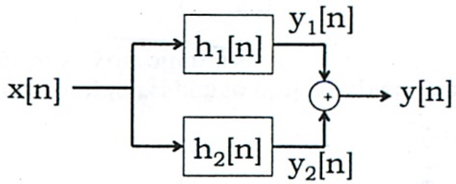
Convolution is associative:

$$x[n]*(h_1[n]*h_2[n])=(x[n]*h_1[n])*h_2[n]$$

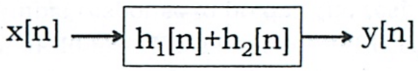
Convolution is distributive:

$$x[n]*(h_1[n]+h_2[n])=x[n]*h_1[n]+x[n]*h_2[n]$$

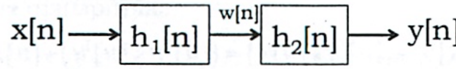
Parallel Interconnection of LTI Systems



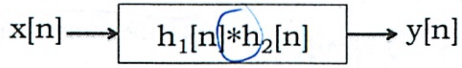
$$y[n] = y_1[n] + y_2[n] = x[n] * h_1[n] + x[n] * h_2[n] = x[n] * (h_1[n] + h_2[n])$$

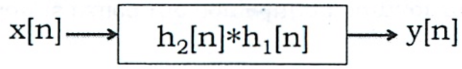


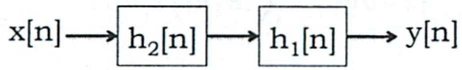
Series Interconnection of LTI Systems



$$y[n] = w[n] * h_2[n] = (x[n] * h_1[n]) * h_2[n] = x[n] * (h_1[n] * h_2[n])$$







Channels as LTI Systems

Many transmission channels can be effectively modeled as LTI systems. When modeling transmissions, there are few simplifications we can make:

- We'll call the time transmissions start t=0; the signal before the start is 0. So x[m] = 0 for m < 0.
- Real-word channels are causal (cause-al): the output at any time depends on values of the input at only the present and past times. So h[m] = 0 for m < 0.

assume input is 0 before transmission starts

non anticipicpant

These two observations allow us to rework the convolution sum when it's used to describe transmission channels:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k] = \sum_{k=0}^{\infty} x[k]h[n-k] = \sum_{k=0}^{n} x[k]h[n-k] = \sum_{j=0}^{n} x[n-j]h[j]$$

start at t=0 causal j=n-k

same products being added together – just differentation

$y[n] = x[n]h[0] + x[n-1]h[1] + x[n-2]h[2] + \dots$

Relationship between h[n] and s[n]

We're often given one of h[n] or s[n] and would like to know the other. On slide #5 we saw

$$\delta[n] = u[n] - u[n-1]$$

Which for LTI systems implies

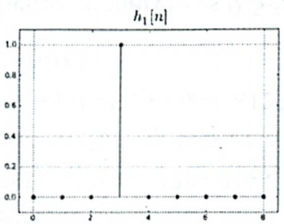
$$h[n] = s[n] - s[n-1]$$

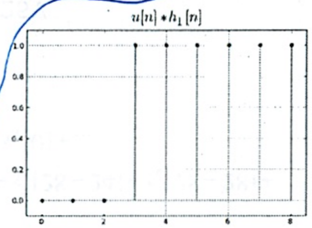
← differences b/w 2 sucessive samples

In other words, the unit sample response is the first difference of the unit step response. Also

$$s[n] = \sum_{k=-\infty}^{n} h[k]$$

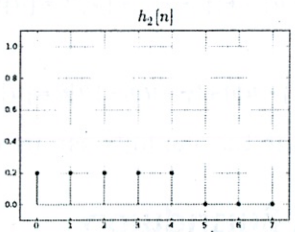
h[n]   s[n]=u[n]*h[n]

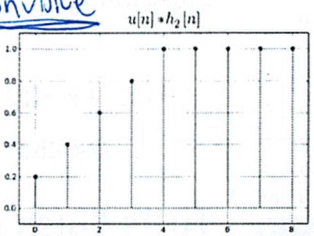




just shift to right 5 samples

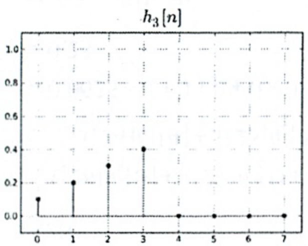
Convolve

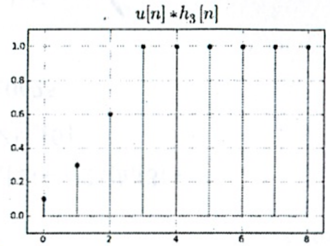




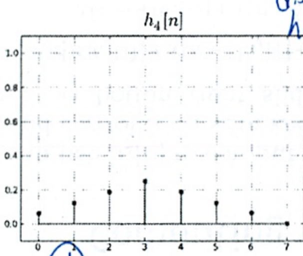
the diff b/w successive values
-differentiation?

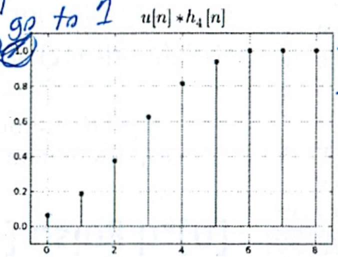
h[n]   s[n]=u[n]*h[n]





does not have to go to 1

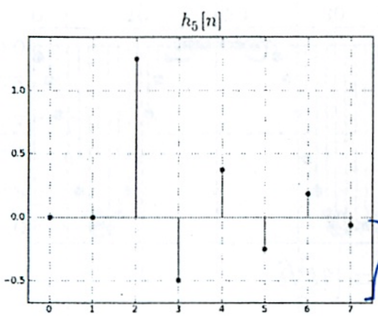




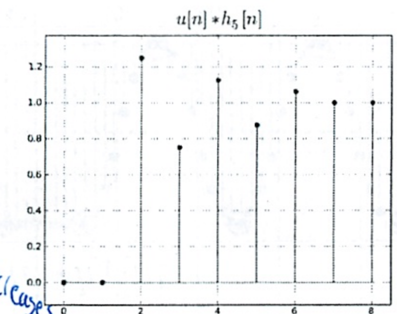
gets less steep?

sum up previous h values

h[n]   s[n]=u[n]*h[n]



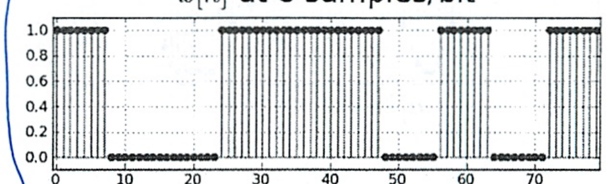
decreases



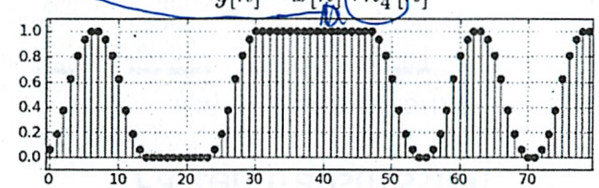
ringing channel

Transmission Over a Channel

$x[n]$ at 8 samples/bit



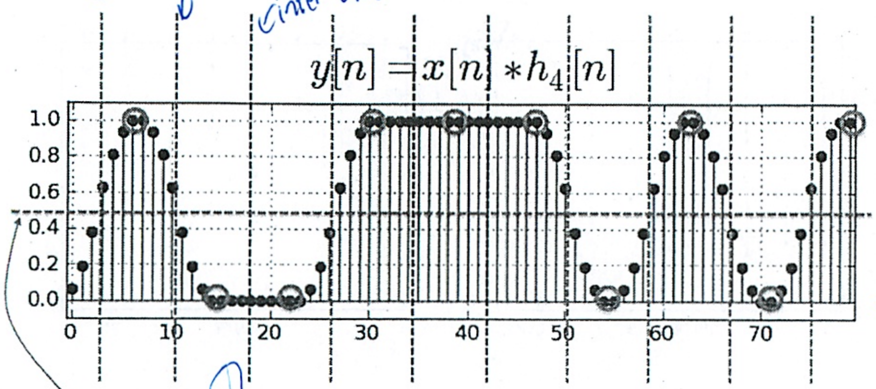
$y[n] = x[n] * h_4[n]$



Receiving the Response

where it draws thresholds

infer others

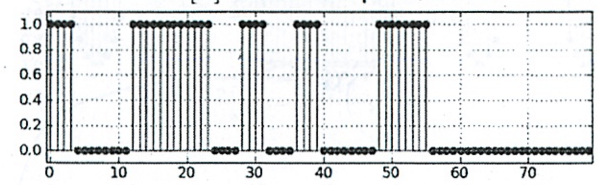


Digitization threshold = 0.5V

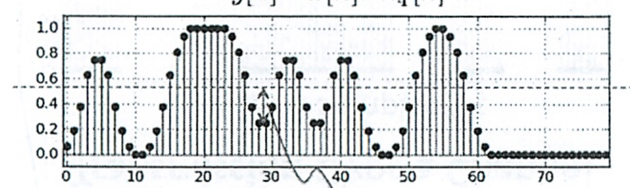
take middle sample

Faster Transmission

$x[n]$ at 4 samples/bit



$y[n] = x[n] * h_4[n]$



Noise margin? $0.5 - y[28]$

more possible error

not enough time for transition

Computing y[28] using $s_4[n]$

n	$s_4[n]$
<0	0.0
0	0.0625
1	0.1875
2	0.375
3	0.625
4	0.8125
5	0.9375
≥6	1.0

We can use $s_4[n]$ to compute y[28] as follows:

$$x[n] = u[n] - u[n-4] + u[n-12] - u[n-24] + u[n-28] + \ldots$$

So

$$y[n] = s_4[n] - s_4[n-4] + s_4[n-12] - s_4[n-24] + s_4[n-28] + \ldots$$

For n=28

$$\begin{aligned} y[28] &= s_4[28] - s_4[28-4] + s_4[28-12] - s_4[28-24] + s_4[28-28] + \ldots \\ &= s_4[28] - s_4[24] + s_4[16] - s_4[4] + s_4[0] + \ldots \\ &= 1.0 - 1.0 + 1.0 - 0.8125 + 0.0625 \\ &= 0.25 \end{aligned}$$

So the noise margin is 0.5-0.25 = 0.25V.

Computing y[28] using $h_4[n]$

n	$h_4[n]$
<0	0.0
0	0.0625
1	0.125
2	0.1875
3	0.25
4	0.1876
5	0.125
6	0.0625
≥7	0.0

We can use $h_4[n]$ to compute y[28] as follows: first expand convolution sum keeping non-zero $h_4[n]$ terms (see bottom right, slide #15):

$$\begin{aligned} y[n] = {} & x[n]h_4[0] + x[n-1]h_4[1] + x[n-2]h_4[2] + x[n-3]h_4[3] + \\ & x[n-4]h_4[4] + x[n-5]h_4[5] + x[n-6]h_4[6] \end{aligned}$$

For n=28:

$$\begin{aligned} y[28] = {} & x[28]h_4[0] + x[27]h_4[1] + x[26]h_4[2] + x[25]h_4[3] + \\ & x[24]h_4[4] + x[23]h_4[5] + x[22]h_4[6] \\ = {} & 0.0625 + 0 + 0 + 0 + 0 + 0.125 + 0.0625 \\ = {} & 0.25 \end{aligned}$$

This agrees with the previous calculation. ✓

Q: example of non linear

what is h_4?

6.02 Recitation

2/15

Why care about linear?

- it lets you do something in a ~~deconvolving~~ decomposing matter

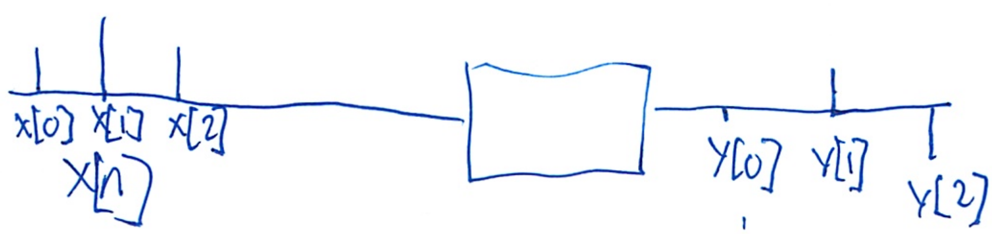
$$f(x_1, x_2) = x_1 + 4x_2$$

$$x_1 \cdot x_2 \quad \leftarrow \text{not linear}$$

- difference w/ your value is always the same
- So you can build stuff in parts
- even if not linear, try to use Talyor Expansion to make it close to linear
- adds up (Maxwell's Eq)
- waves naturally are linear - add up

Linear Time-Invarient Systems (LTI)

- in 90s plenty of ultimatly false theories about non LTI systems
  - basically non linear (see above)



Causalty

This is a function of all things inserted before + noise

2

There is nothing special at time 0

- its not different on a Tuesday ...
- when ever the channel has been zeroed out

time invarient

$$\tilde{x}[n] = x[n-7] \quad \longrightarrow \quad \tilde{y}[n] = y[n-7]$$

$\swarrow$ tilde

$$\begin{bmatrix} y[0] \\ \vdots \\ y[T] \end{bmatrix} = F \begin{bmatrix} x[0] \\ \vdots \\ y[T] \end{bmatrix}$$

$$y[0] \overset{?}{=} F_0 \left(\begin{bmatrix} x[0] \\ \vdots \\ x[T] \end{bmatrix} \right)$$

F_0 can depend only on $x[0]$

F_1 " " " " $x[0]$ and $x[1]$

F_t " " " " $x[0]$, $x[t+1]$, ... $x[t]$

$$y[t] = F_T\left(x[0], \ldots, x[T] \right) =$$

example $\quad x[0]^2 + x[1] * x[2] + x[T]' x[4] x[9] \ldots$ $\leftarrow$ non linear function

(3)

Linear fn $= \alpha_T x[T] + \alpha_{T-1} x[T-1] + \dots + \alpha_0 x[0]$

$$
\begin{bmatrix} y[0] \\ y[1] \\ \vdots \\ y[T] \end{bmatrix}
=
\begin{bmatrix}
\alpha_0 & 0 & 0 & \dots & 0 \\
\alpha_0 & \alpha_1 & 0 & \dots & 0 \\
\alpha_0 & \alpha_1 & \alpha_2 & 0 \dots & 0 \\
\vdots & & & \vdots & \vdots \\
\alpha_0 & \alpha_1 & \dots & & \alpha_T
\end{bmatrix}
\begin{bmatrix} x[0] \\ x[1] \\ \vdots \\ x[T] \end{bmatrix}
$$

not a function of time
since is time invarient

Say only depends on last 7 symbols

$$
\begin{bmatrix} \vdots \\ y[T] \\ \vdots \end{bmatrix}
=
\begin{bmatrix}
\ddots & & & & & & \\
h_6 \dots & h_1 & h_0 & 0 & 0 & \dots & 0 \\
& & & & & & \ddots
\end{bmatrix}
\begin{bmatrix} \vdots \\ x[T-6] \\ \vdots \\ x[T-2] \\ x[T-1] \\ x[T] \\ \vdots \end{bmatrix}
$$

4

(He is doing a very complex way to explain)
(Its same as 6.01)   └ very abstract way

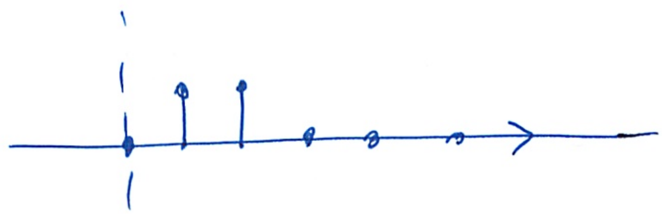
Problems

Input $x[0] = 0$

$x[1] = 1$

$x[2] = 1$

$x[n] = 0$ ~~for~~ $n \geq 3$



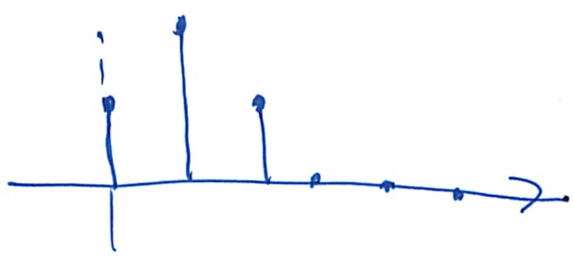
Output

$y[0] = 1$

$y[1] = 2$

$y[2] = 1$

$y[n] < 0$ $n \geq 3$



Is the system causal?

No - $y[0]$ should be 0 since $x[0] = 0$

What is the output when input changes? add

$\tilde{x} = x[3] = 1$

$\tilde{x} = x[4] = 1$ $\tilde{x} = x$ (keep rest of rules)

↑ x tilde (like x prime I think)

(5) Could also say

$\tilde{x}[n] = x[n] + x[n-2]$ for all n

So output is

$\tilde{y}[n] = y[n] + y[n-2]$

$\tilde{y}[0] = 1 = 1 + 0$

$\tilde{y}[1] = 2 = 2 + 0$

$\tilde{y}[2] = 2 = 1 + 1$

$\tilde{y}[3] = 2 = 0 + 2$

$\tilde{y}[4] = 1 = 0 + 1$

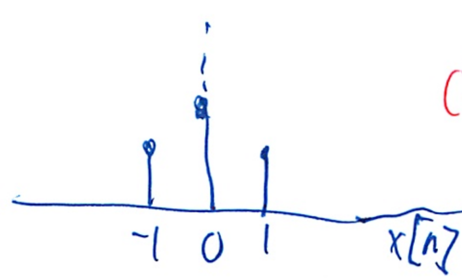
$\tilde{y}[5] = 0 = 0 + 0$

Thats interesting that can use output chart and just shift that

(Don't think we did that last year)

Sample Response

$$h[n] = \begin{cases} 1 & n = -1 \\ 2 & n = 0 \\ 1 & n = 1 \\ 0 & \text{otherwise} \quad n \le -2,\ n \ge 2 \end{cases}$$



Charistics of wire

"fading Coefficents"

(6)

$$y[n] = h_0\, x[n]$$

$$= h_1\, x[n-1] + h_2\, x[n-2] + \dots + h_k\, x[n-k]$$

$$= h_{-1}\, x[n+1] + h_{-2}\, x[n+2] + \dots$$

$$x[n] = \begin{cases} 1 & n=0 \\ 2 & n=1 \\ 3 & n=2 \\ 0 & \text{otherwise} \quad n<0,\ n \geq 3 \end{cases}$$

The signal

$$y[0] = h_0\, x[0] + h_1\, x[-1] + h_{-1}\, x[1]$$

$$= 2 \cdot 1 + 1 \cdot 0 + 1 \cdot 2$$

$$= 4$$

$$y[1] = \dots$$

$$y[2] = h_0 x[2] + h_1\, x[2-1] + h_{-1}\, x[3]$$

$$= 2 \cdot 3 + 1 \cdot 2 + 1 \cdot 0$$

$$= 8$$

Channel is not causal