- **Review and expansion on previous class**
  - Can we compress all strings?
  - Huffman codes (grouping and choosing between codes with equal average length).
  - Integer implementation of arithmetic coding
  - Conditional entropy and Markov models
  - The JBIG standard
  - The use of Tries is LZW and space issues
  - Run length coding (used for Fax coding) and residual compression (used in the lossless JPEG standard)
- **Lossy compression**
  - Intro
  - Scalar quantization
  - Vector quantization
  - Intro to transform coding

1 Lossless Compression: review and expansion

1.1 Can we compress all strings?

Can a compression algorithm compress some messages without expanding others?

Not if all strings are valid inputs

For a character set of size m and messages of length n, if one string is compressed to $l < n \cdot log_2(m)$, then at least one string must expand to $l > n \cdot log_2(m)$.

GZIP will only expand slightly if no compression is possible. Alternatively, GZIP might check to see if no compression occurred.

1.2 Huffman Codes Revisited

It can be shown that $H(s) \leq R < H(s) + 1$ where $H(s)$ is entropy and R is the average code length. Proof of this inequality will be a homework problem. The difference $H(s) - R$ is the overhead caused by Huffman coding.

A standard trick to reduce the overhead per-symbol is to group symbols. For example, if there are 6 symbols, we can generate the probabilities for all 36 pairs by multiplying the probabilities of the symbols (there will be at most 21 unique probabilities). Now generate a

Huffman code for these probabilities and use it to code two symbols at a time. Note that we are not taking advantage of patterns among characters as done with the LZ algorithms or Conditional Entropy discussed later (we are assuming the probability of each symbol is independent of others). Since the overhead is now distributed across pairs of symbols the average per-symbol overhead is bounded by 1/2. If we grouped 3 symbols it would be 1/3. This works fine for a few symbols, but the table of probabilities and the number of entries in the Huffman tree will quickly grow large when increasing the number of symbols in the group.

All Huffman codes for a given set of probabilities will have the same length, but there are different possible ways of encoding.

symbol	probability	code 1	code 2
a	0.2	01	10
b	0.4	1	00
c	0.2	000	11
d	0.1	0010	010
e	0.1	0011	011

Both codings produce an average of 2.2 bits per symbol

With these two codings, one has a high variance in length (code 1), while the other has a low variance (code 2). With low variance coding, a constant character transmission rate is more easily achieved. Also, buffer requirements in a low variance coding scheme might be less. Lowest variance coding can be guaranteed by the following construction.

- After joining two nodes with the lowest probabilities (highest priority), give this joint probability a lower priority than other existing nodes with the same probability.
- In the example above, after 'd' and 'e' are joined, the pair will have the same probability as 'c' and 'a' (.2), but is given a lower probability so that 'c' and 'a' are joined next.
- Now, 'ac' has the lowest priority in the group of 0.4 probabilities
- 'b' and 'de' are joined, and finally, 'ac' is joined with 'bde' revealing the tree shown in Figure 1

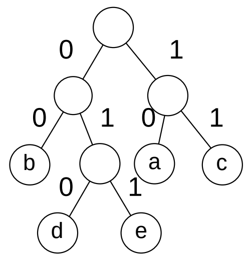

Figure 1: Binary tree for Huffman code 2

1.3 Arithmetic Coding Revisited

It can be shown that Arithmetic coding is bounded as follows: $m \cdot H(s) \leq l \leq m \cdot H(s) + 2$ where m is the length of the input, l is the length of the output, and H(s) is the per-symbol entropy.

An efficient and easy way of implementing arithmetic coding is by using the following integer implementation. The implementation won't give exact arithmetic coding because of roundoff error, but will be close. If both sender and receiver use the same arithmetic they will round in the same way making it possible for the receiver to decode the message properly.

The Integer Implementation is as follows:

- Assume probabilities are counts $c_1, c_2, \ldots, c_m$ (Fractions will also work)
- Cumulative Count: $C_i = \sum_{j=1}^{i} c_j$
- Total Count: $T = \sum_{j=1}^{m} c_j$
- As with any arithmetic coding we start with a fixed-sized region and narrow it each time we code a new character. In our integer implementation the region is initialized to $[0..(R-1)]$ where $R = 2^k$ (i.e. is a power of 2).
- To avoid narrowing to a region of size less than 1, we scale (expand) the region by a factor of 2 under the following conditions:
  - If the region is narrowed down such that it is either in the top or bottom half of the space, output a 1 or 0, respectively, and scale using the appropriate scaling rule.
  - If the region is narrowed down to the second and third quarters, then the region will be expanded about the middle using scale rule #3, described below.
- The Algorithm
  - Let u and l be integers $< K = 2^k$
  - Initial values: $u_0 = K - 1; l_0 = 0$
  - $u_j = l_{j-1} + \left\lfloor (u_{j-1} - l_{j-1} + 1)\frac{C_{x_j}}{T} \right\rfloor - 1$
  - $l_j = l_{j-1} + \left\lfloor (u_{j-1} - l_{j-1} + 1)\frac{C_{x_j - 1}}{T} \right\rfloor$
- Scaling Rules
  - **#1:** If $u_j = 1\ldots$ and $l_j = 1\ldots$ then (the region of interest is in the top half)
    output 1
    shift u to the left and insert a 1
    shift l to the left and insert a 0
    output s zeros, where s is the number of times expanded about the middle region
    set s = 0

- **#2:** If $u_j = 0\ldots$ and $l_j = 0\ldots$ then (the region of interest is in the bottom half)
  output 0
  shift u to the left and insert a 1
  shift l to the left and insert a 0
  output s ones, where s is the number of times expanded about the middle region
  set s = 0
- **#3:** If $u_j = 10\ldots$ and $l_j = 01\ldots$ then (the region of interest is in the second and third quarters of the range)
  $s = s + 1$
  shift u to the left and insert a 1
  shift l to the left and insert a 0
  complement the MSB (most significant bit) of l and u

Question:

If we expand about the middle, expand about the middle, and expand in the top, what output do we get?

Following the rules, s gets incremented twice by expanding about the middle. When we expand in the top half, we output a 1 and s number of zeros (in this case, s=2). Output: 100

Example:

Let $c_1 = 40, c_2 = 1, c_3 = 9$

Which gives $C_0 = 0, C_1 = 40, C_2 = 41, C_3 = T = 50$

We chose a k such that $4 \cdot T < K = 2^k$

In this case, $T = 50$, so $k = 8, K = 256$.

The message we want to transmit is 1321

i	u_i	binary	l_i	binary	output	scale rule	x_i	C_{x_i-1}	C_{x_i}
0	255	11111111	0	00000000			1	0	40
1	203	11001011	0	00000000			3	41	50
2	203	11001011	167	10100111	1	Rule 1			
2	151	10010111	78	01001110		Rule 3			
2	175	10101111	28	00011100			2	40	41
3	148	10010100	146	10010010	10	Rule 1			
3	41	00101001	36	00100100	0	Rule 2			
3	83	01010011	72	01001000	0	Rule 2			
3	167	10100111	144	10010000	1	Rule 1			
3	79	01001111	32	00100000	0	Rule 2			
3	159	10011111	64	01000000		Rule 3			
3	191	10111111	0	00000000			1	0	40
4	152	10011000	0	00000000					

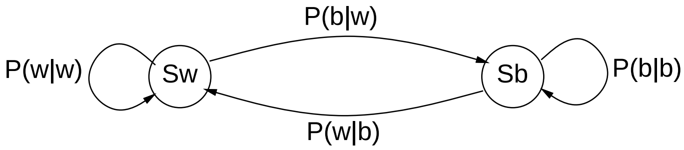

Figure 2: A two state Markov Model

1.4 Conditional Entropy and Markov Chains

Conditional entropy comes into use in both lossless and lossy techniques. Compression can be increased by knowing the context of the message (symbol). For example, in images, the surrounding pixels can be used as the context. In black and white faxes, if there is a black pixel, and the surrounding areas also contain black pixels, it is highly probable that the next pixel is also black. For text, previous characters can be used as the context.

The *conditional entropy* of T based on context S is defined by the equation:

$$\begin{aligned} H(T|S) &= \sum_j P(S_j) \sum_i P(T_i|S_j) log \frac{1}{P(T_i|S_j)} \\ where & \\ P(S_j) &= probability\ of\ context\ S_j \\ P(T_i|S_j) &= probability\ of\ T_i\ in\ context\ S_j \end{aligned}$$

It is not hard to show that if T is independent of the context S then $H(T|S) = H(T)$ otherwise $H(T|S) < H(T)$. In other words, knowing the context can only reduce the entropy.

A way to represent these conditional probabilities is through a Markov Model (or Markov Chain). Figure 2 shows a Markov Model where there are two states, white (Sw) and black (Sb), and conditional probabilities:

$$\begin{aligned} P(w|w) &= probability\ of\ getting\ a\ white\ pixel\ if\ the\ previous\ pixel\ was\ white \\ P(w|b) &= probability\ of\ getting\ a\ white\ pixel\ if\ the\ previous\ pixel\ was\ black \\ P(b|b) &= probability\ of\ getting\ a\ black\ pixel\ if\ the\ previous\ pixel\ was\ black \\ P(b|w) &= probability\ of\ getting\ a\ black\ pixel\ if\ the\ previous\ pixel\ was\ white \end{aligned}$$

In the case of a fax machine, the probabilities of staying in the same state, $P(b|b)$ and $P(w|w)$, are higher than the probability of switching states.

Conditional probabilities can be used with arithmetic coding to code messages with average length that asymptotically matches the conditional entropy. The idea is that for every context we have a separate narrowing function based on the probabilities for that context.

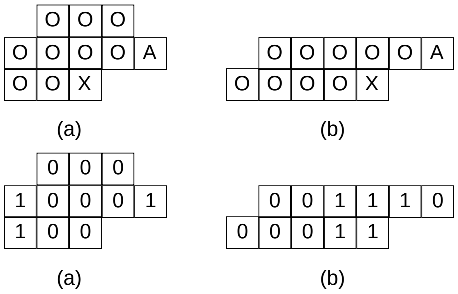

Figure 3: a) Three-line neighborhood b) Two-line neighborhood

1.5 The JBIG Standard

JBIG stands for the Joint Bilevel Image Processing Group. Probabilities are generated based on context. This standard is used for black and white, where the table of probabilities is only 2^k, where k is the number of pixels used as the context. Neighborhoods are used as the context. Example neighborhoods are shown in Figure 3. In the JBIG standard, there is also what is known as a roaming pixel. One of the context pixels, 'A' in Figure 3, is possibly located n pixels away. The current pixel is denoted by 'X'. This is useful when there are vertical lines being transmitted. If the JBIG standard was used on color images, then the table would grow too large to be practical. After building a probability table, the image is then sent after being arithmetically encoded. There is also a default probability table that was made after analyzing thousands of images.

1.6 Tries and use in LZW

An example trie structure is shown in Figure 4. When using a trie structure for LZW, searches can be done in unit time, since the previous search is the parent of the next search. The problem with using tries is that the amount of memory needed for a trie can get large. Consider a dictionary of size 4096. The pointers in the trie need to be $log_2(4096)$ bits = 1.5 bytes long. If we have 256 symbols the amount of memory we will need is 4096 entries * 256 child pointers * 1.5 bytes/pointer $\approx$ 1.5 Megabytes!

Another way of storing the dictionary is by using hash tables. The choice of whether to use a hash table or a trie structure depends mostly on the size of the dictionary and the available memory.

1.7 Other Lossless Compression Techniques

In older fax machines, run length compression is used to transmit and receive messages. The algorithm creates a list of numbers that correspond to the number of consecutive black dots then the number of consecutive white dots and repeats until the end of the message. The list is then Huffman coded and transmitted to the receiving fax, which decodes by applying the algorithm in reverse. This method works OK for black and white images, but JBIG is two to three times better.

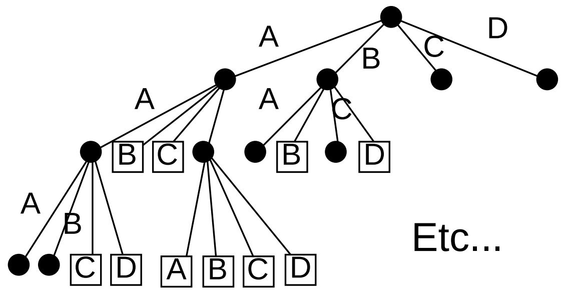

Figure 4: A trie structure

Residual compression is another lossless compression technique. It is used in lossless JPEG, which handles both grey-scale and color images. Residual compression guesses the next pixel value based on context and outputs the difference between the guess and actual value. The differences are Huffman coded and transmitted. Guesses of the next pixel are usually a weighted sum of pixels in the neighborhood. For the most part, the guesses are good. Instead of Huffman coding the original pixel values we now Huffman code (or arithmetic code) the residuals. The residuals will tend to have a lower entropy than the original data (i.e. 0s and low values have high probabilities, while high values have low probabilities) giving us a more efficient coding. Note that the residual values can be negative.

2 Lossy Compression

There are many ways to achieve lossy compression. Some of these include

- scalar quantization,
- vector quantization,
- transform coding (wavelets, block cosine in JPEG and MPEG),
- fractal coding, and model based coding.

Judging the effectiveness of lossy compression is difficult. Unlike lossless compression where measures are easily quantifiable (compression ratio, time, etc.), lossy compression also has some qualitative measures. In fact many people argue that the quantitative measures for lossy compression such as root-mean-square error are not good measures of human perception.

2.1 Scalar Quantization

This is the simplest technique. The idea is to group a large number of regions to a smaller number of regions. An example of this is using only the most significant 6 bits of an 8 bit message. This is an example of a uniform scalar quantization. The function is shown in Figure 5a.

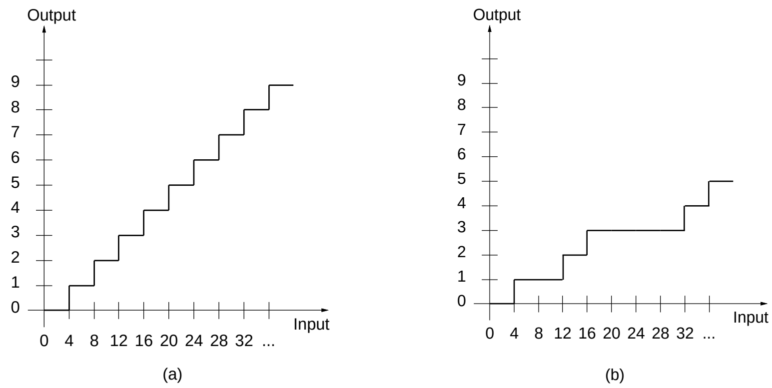

Figure 5: a) Uniform Scalar Quantization b) Nonuniform Scalar Quantization

Non-uniform scalar quantizations might have a function similar to Figure 5b. Non-uniform quantizations are useful when some regions have more useful information than others. For example, differences in low values of red are more sensitive to the eyes than differences in high values of red. In this case, keeping as much information when the values are low is important, while grouping some of the high values will lead to a very good approximation. For images, this allows for better quality for the same compression.

2.2 Vector Quantization

The idea of vector quantization is given a set S of possible vectors to generate a representative subset $R \subset S$ which can be used to approximate the elements of S. The set R are called the code vectors. The incoming data gets parsed into vectors and the encoder first finds the closest (by Euclidean norms) code vector. The index of the code vector can now be sent (possibly after the normal Huffman coding). The quality of this compression will depend on the size of the codebook. Figure 6 shows a block diagram of vector quantization.

What should be used as the vectors? In images, the values of red, green, and blue can be used as a vector. Vector quantization can be used to reduce 24 bit color to 8 bit color. Many terminals use some sort of vector quantization to save on screen memory, and it is supported by the graphics hardware. For speech, a pre-defined number, K, of consecutive samples can be used as the vector. Another way to break up an image into vectors is to represent blocks of pixels as a vector.

Vector quantization used along with a residual method produces a lossless compression. All that needs to be transmitted is the index of the code vector and the difference between the code vector and the actual vector.

2.3 Transform Coding

To use transform coding, a linear set of basis functions (ϕ_i) that span the space must first be selected. Some common sets include sin, cos, spherical harmonics, Bessel functions, wavelets, and rotation. Figure 7 shows some examples of the first three basis functions for

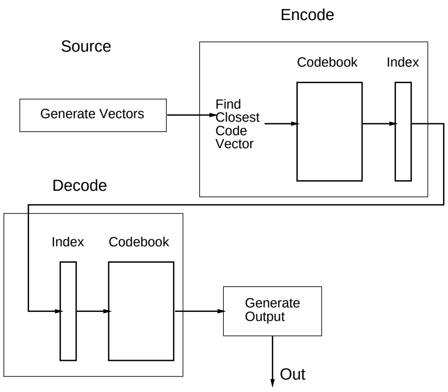

Figure 6: Vector Quantization

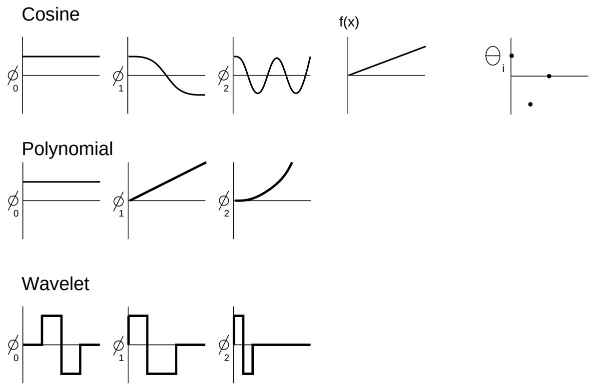

Figure 7: Transforms

cosine, polynomial, and wavelet transformations. The wavelet transform is especially good at capturing local features.

The general equation for generating the i^{th} transform coefficient Θ_i given a function $F(x)$ to transform is

$$\begin{aligned} \Theta_i &= \sum_{j=1}^{n} \phi_i(x_j) F(x_j) \\ where\ \Theta_i &= i^{th}\ transformed\ coefficient \end{aligned}$$

2.4 Choosing a Transform

The goals we have when choosing a transform are:

- Decorrelate the data
- Low coefficients for many terms
- Some basis functions can be ignored by perception

Low coefficients for many terms will allow us to either drop them and only send the larger coefficients. The higher frequency cosines will not be able to be perceived, so it is allowable to drop them. This is low pass filtering a signal (taking only the lower frequency components of a FFT).

Question:

Why not use a Fourier (or cosine) transform across the whole image?

Imagine a totally white background with a black pixel in the middle. This is represented as an impulse function, and the Fourier transform of an impulse function is 1 for all frequencies. To adequately represent the image, we will not be able to drop any coefficients since they are all of the same magnitude. JPEG blocks the image off before applying transforms.