More Tidbits on Data Compression

(Above, Lena, unwitting data compression spokeswoman)

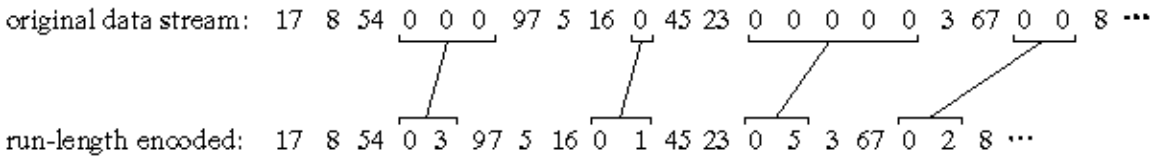
We have already discussed creating Huffman codes. Let's examine some other methods of compressing data (run-length, arithmetic, and LZW) and also look briefly at the format used in JPEG images.

Run-Length Encoding

Data files often contain the same character repeated in sequence. For example, text files contain multiple spaces for formatting tables or columns. Images may contain multiple 0's or 1's for all black or all white. Digitized signals often contain long runs of zeros, for example quiet time between notes or times when the signal is not changing.

The idea behind run length encoding is to represent long runs of zeros. Each run of zeros (or some other repeating character) is represented by a flag indicating that run-length compression is beginning. The flag is followed by the number of zeros. The flag could be expanded to include the actual repeating character if we wish to use this technique for more than just zeros.

The process on a sample datastream is indicated below:



The PackBits program on the Macintosh used a generalized RLE scheme for data compression.

Arithmetic Encoding

In arithmetic encoding, we turn an entire datastream into a single number! The more data we have, the greater precision we will need in representing the number.

There are two fundamentals in arithmetic coding: the probability of a symbol and its encoding interval range. The probabilities of source symbols determine the compression efficiency. They also determine the interval ranges of source symbols for the encoding process. These interval ranges are contained within the interval from zero to one. The interval ranges for the encoding process determine the compression output. This is best demonstrated by an example.

Let us assume that the source symbols are { 00, 01, 10, 11 } and the probabilities of these symbols are { 0.1, 0.4, 0.2, 0.3 }, respectively. Then, based on these probabilities, the interval [0,1) can be divided as four sub-intervals: [0,0.1), [0.1,0.5), [0.5,0.7), [0.7,1), where [x,y) denotes a half open interval, which includes x but excludes y. The above information can be summarized in table:

Symbols	00	01	10	11
Probabilities	0.1	0.4	0.2	0.3

Initial Encoding Intervals	[0,0.1)	[0.1,0.5)	[0.5,0.7)	[0.7,1)

To encode a message of a binary sequence 10 00 11 00:

We take the first symbol 10 from the message and find its encoding range is [0.5,0.7). Since the range of the second symbol 00 from the message is [0,1), it is encoded by taking the first 10th of interval [0.5,0.7) as the new interval [0.5,0.52). In other words, we treat the range from [0.5, 0.7) as an entire interval, and then use the first 0.1 of this interval for the second symbol. This process is repeated for the rest of the symbols. To encode the third symbol 11, we have a new interval [0.514,0.52). After encoding the fourth symbol 00, the new interval is [0.514,0.5146). The compression output of this message can be any number in the last interval. For example, we could pick 0.5145.

To decode the data, we apply the process in reverse. We need to know the probability ranges, presumably transmitted with the data unencoded:

Given 0.5145, the value is in the range [0.5, 0.7) so it is symbol 10.
Given 0.5145, the value is in the 1^{st} 10^{th} of the interval [0.5, 0.7) so it is symbol 00.
Given 0.5145, the value is in the 7^{th} 10^{th} of the interval [0.5, 0.52) so it is symbol 11.
Given 0.5145, the value is in the 1^{st} 10^{th} of the interval [0.514, 0.52) so it is symbol 00.

The resulting sequence of symbols is now 10, 00, 11, 00.

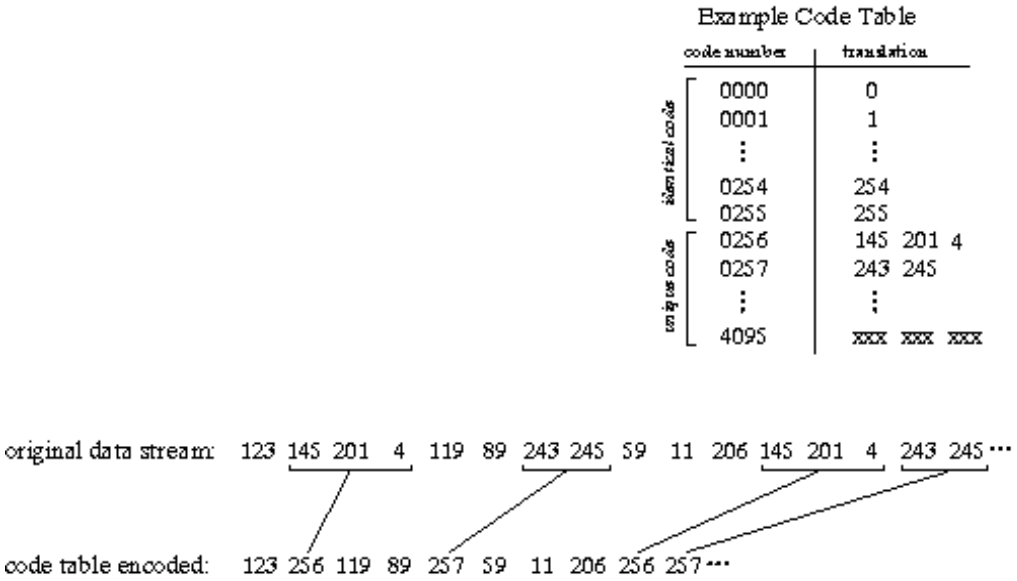
Some issues to note:

1) Since no single machine exists with an infinite precision, "underflow" and "overflow" are the obvious problems for the real world machines. We can use multiple bytes or words to represent higher precision if necessary, at additional expense in decoding these bytes and mapping them to the precision of the machine (e.g. 32, 64 bits).
2) An arithmetic coder produces only one codeword, a real number in interval [0,1), for the entire message to be transmitted. We cannot perform decoding process until we received all bits representing this real number.
3) Arithmetic coding is an error sensitive compression scheme. A single bit error can corrupt the entire message.

LZW Compression

This is a very popular form of compression that is used as the basis for many commercial and non-commercial compression programs (gzip, pkzip, GIF, compressed postscript, “disk doublers”). LZW compression is named after its developers, A. Lempel and J. Ziv, with later modifications by Terry A. Welch. It is the foremost technique for general purpose data compression due to its simplicity and versatility. Typically, you can expect LZW to compress text, executable code, and similar data files to about one-half their original size.

LZW is similar to Huffman encoding, except it uses a code table for sequences of characters. Consider the example below:



LZW compression uses a code table. A common choice is to provide 4096 entries in the table. In this case, the LZW encoded data consists entirely of 12 bit codes, each referring to one of the entries in the code table. Uncompression is achieved by taking each code from the compressed file, and translating it through the code table to find what character or characters it represents. Codes 0-255 in the code table are always assigned to represent single bytes from the input file. For example, if only these first 256 codes were used, each byte in the original file would be converted into 12 bits in the LZW encoded file, resulting in a 50% larger file size. During uncompression, each 12 bit code would be translated via the code table back into the single bytes. Of course, this wouldn't be a useful situation.

The LZW method achieves compression by using codes 256 through 4095 to represent sequences of bytes. For example, code 523 may represent the sequence of three bytes: 231 124 234. Each time the compression algorithm encounters this sequence in the input file, code 523 is placed in the encoded file. During uncompression, code 523 is translated via the code table to recreate the true 3 byte sequence. The longer the sequence assigned to a single code, and the more often the sequence is repeated, the higher the compression achieved.

Although this is a simple approach, there are two major obstacles that need to be overcome: (1) how to determine what sequences should be in the code table, and (2) how to efficiently determine if a sequence of data has an entry in the code table.

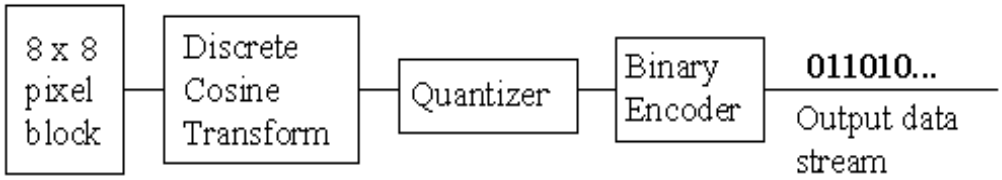
#1 is the subject of many algorithms (the authors of LZW propose one way to determine the codes using a state table). For example, one could devise any algorithm to find repeat strings and use that to determine the code table sequences. The longest, most repeating strings could be given a code assignment.

#2 could be implemented in a simple or complicated manner. However this implementation will greatly affect the execution time of the compression algorithm because it must search the code table to determine if a match is present. As an analogy, imagine you want to find if a friend's name is listed in the telephone directory. The catch is, the only directory you have is arranged by telephone number, not alphabetical order. This requires you to search page after page trying to find the name you want. This inefficient situation is exactly the same as searching all 4096 codes for a match to a specific character string. The answer: organize the code table so that what you are looking for tells you where to look (like a partially alphabetized telephone directory). In other words, don't assign the 4096 codes to sequential locations in memory. Rather, divide the memory into sections based on what sequences will be stored there. For example, suppose we

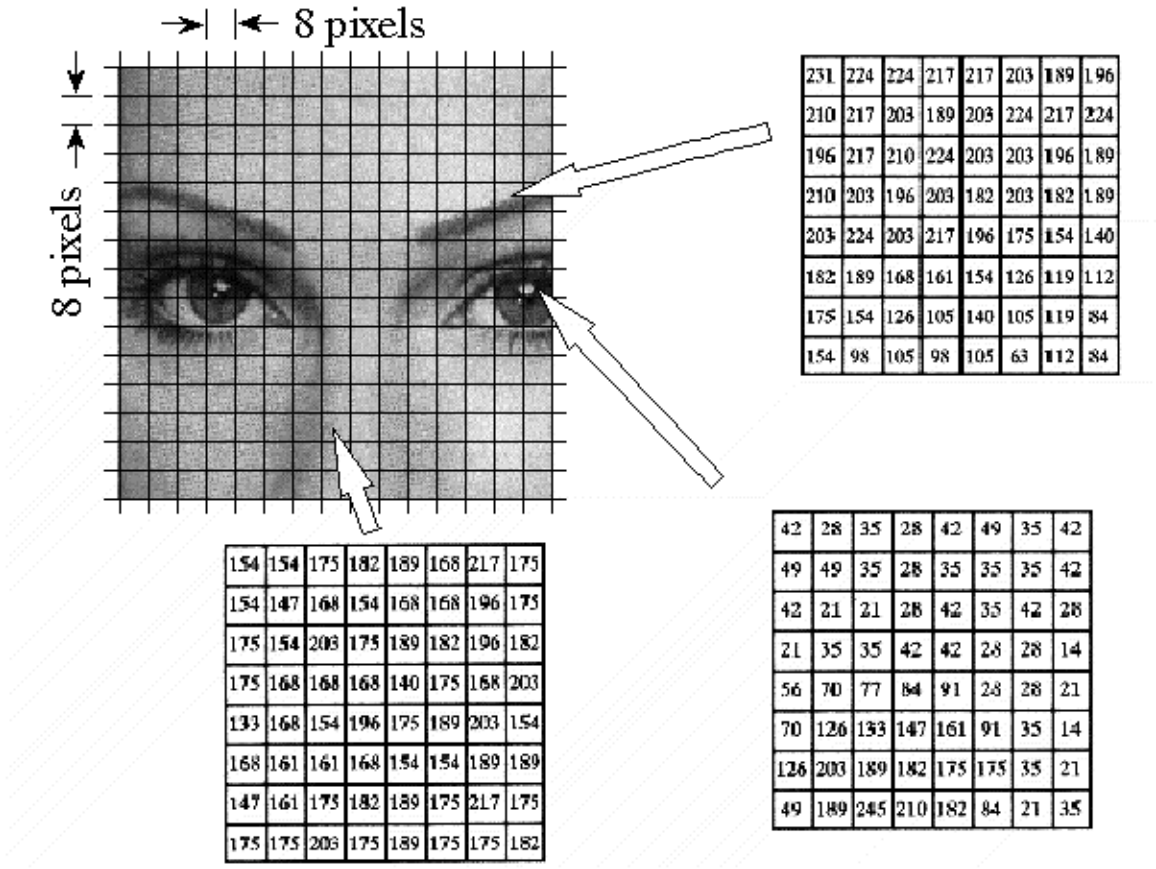
want to find if the sequence: code 329 + x, is in the code table. The code table should be organized so that the "x" indicates where to starting looking. There are other schemes using hash tables, additional indices, and other techniques to help speed up the lookup process. There are many schemes for this type of code table management, and they can become quite complicated.

JPEG Image Compression

Many methods of lossy compression have been developed; however, a family of techniques called transform compression has proven the most valuable. The most popular for images, and one of the best techniques, is the JPEG format. JPEG is named after its origin, the Joint Photographers Experts Group. We will describe the general operation of JPEG to illustrate how lossy compression works. A block diagram of the encoding process is below.



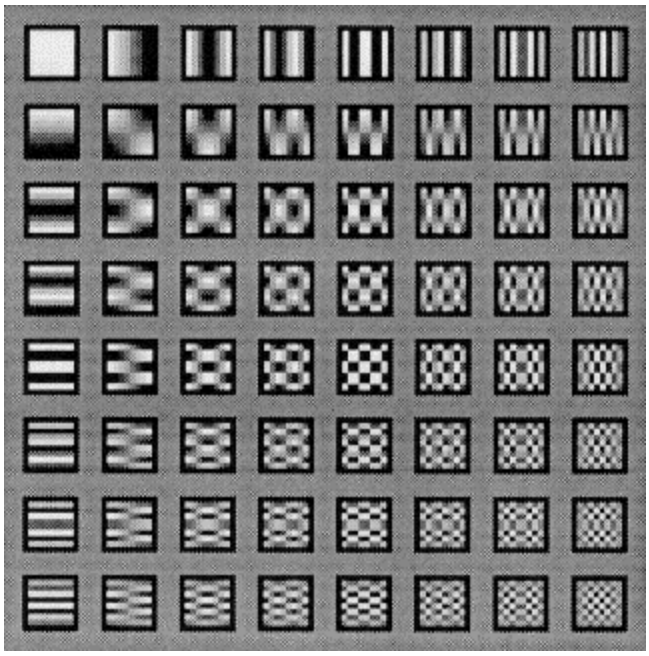
JPEG compression starts by breaking the image into 8x8 pixel groups. The full JPEG algorithm can accept a wide range of bits per pixel, including the use of color information. In this example, each pixel is a single byte, a grayscale value between 0 and 255. These 8x8 pixel groups are treated independently during compression. That is, each group is initially represented by 64 bytes. Why use 8x8 pixel groups instead of, for instance, 16x16? The 8x8 grouping was based on the maximum size that integrated circuit technology could handle at the time the standard was developed. In any event, the 8x8 size works well, and it may or may not be changed in the future.



The 8x8 group idea is demonstrated above on a greyscale image. If we were working with a color image, we would essentially have the same picture for a Red, Green, and Blue component.

The next idea is to treat each 8x8 group as a signal in the frequency domain. For example, in the Fourier transform, we can represent a signal as a summation of sine and cosine waves. JPEG uses the Discrete Cosine Transform (DCT), a relative of the Fourier transform, but the DCT only uses cosine waves.

When the DCT is taken of an 8x8 group, it results in an 8x8 spectrum. In other words, 64 numbers are changed into 64 other numbers. All these values are real; there is no complex mathematics here. Just as in Fourier analysis, each value in the spectrum is the amplitude of a basis function. You can think of these numbers as coefficients. Each coefficient represents the intensity of a particular 8x8 image each of which is shown in the diagram below. These 64 coefficients can be combined to give back the original 8x8 block of the image to be compressed.

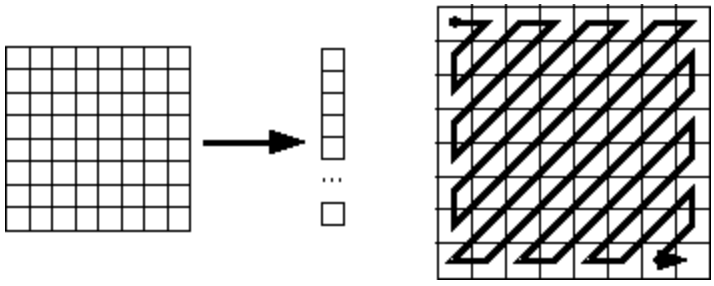


This is a visual representation of 64 images that result from the DCT. The images in the upper left correspond to "high frequency" spatial images, while the lower right are lower frequency. We can re-create any 8x8 image block using these basis/principal-component images. For example, 255*(upper-left-image)+0*(all other images) results in an all white block of pixels.

This at first, does not appear to give any saving since each DCT coefficient is represented with with 8 bits. The saving comes into play when you consider that the high spatial frequency DCT coefficients occur less and actually make less visual impact on the image. This means these DCT coefficients can be represented with fewer bits (ie. they are less important to the image).

This step of reducing the number of bits representing DCT coefficients is called quantization. For each JPEG compressed image, there is a quantization table that determines how many bits represent each DCT coefficient. By using fewer bits, we get better compression.

First, when you consider how to order the DCT coefficients, it would be reasonable to order them in a zig-zag way as shown in the diagram below:



Given that the high spatial frequency DCT coefficients are less likely to be needed, many of these values will be zero and so improve compression when using Run-Length, Huffman or Arithmetic Encoding.

In the JPEG compression so far, we have 64 DCT coefficients each of which can have values to differing degrees of accuracy. This gives rise to the fact that many of the coefficients will be zero valued and so using Variable Length Integers (VLI) would improve the compression. Essentially, we determine in advance which image blocks should be represented with high precision (8 bits) or low precision (1 or 2 bits). High

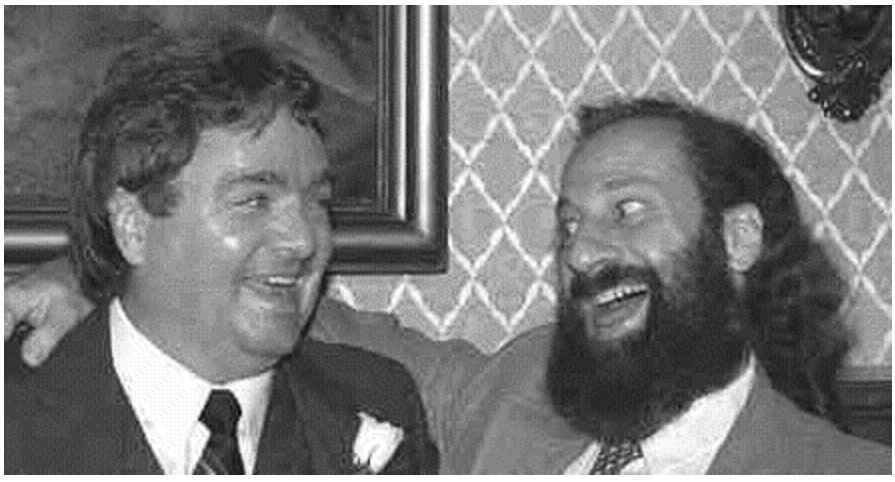
frequencies are not as visible to humans for example, so it makes sense to only use a couple of bits to represent these values, resulting in compression with almost no perceptible difference to humans. Since some of the run-size codes occur more frequently than others, it would be prudent to use either Huffman or Arithmetic Encoding. JPEG uses Huffman coding even though Arithmetic encoding gives 5-10% improvement over Huffman, this is because IBM own the intellectual property on Arithmetic Encoding and JPEG was designed to be freely available.

Note that we can use the quantization step to vary the amount of compression. If we only use a couple of bits to represent each coefficient, then we will have high compression at the cost of a fuzzy image. Similarly, we could use all the bits (but compressed) for an exact replica of the original image.

Here are a couple of images with varying quality factors from an original 82K

QF 75 21K

QF 20 7.5K



QF 5 3.2K

QF3 2.5K