Compressing Information

How much information is contained in an ASCII file giving the schedule of Amtrak trains between Boston and New York? A random sequence of one million bytes? A CD recorded by Britney Spears's younger sister?

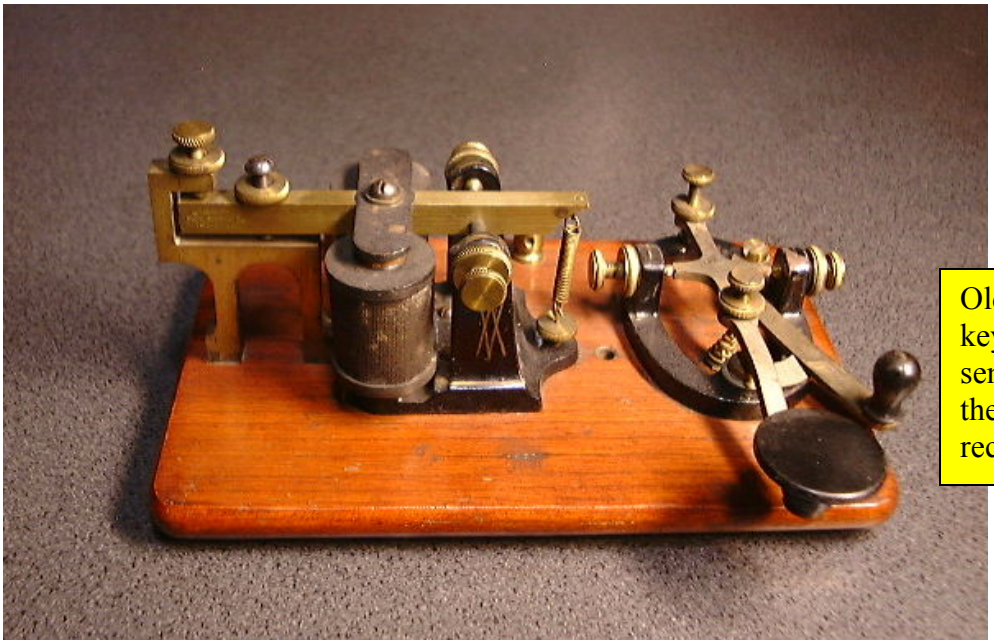
There is a precise way of answering this kind of question, and it has nothing to do with the subjective question of *how informative* the file is. We take the question to mean, how many bits do we have to save in order to be able to recover all the bits of the original file?

If the original file contained audio, or an image, we may not be worried about recovering all the bits of the original file, we just want whatever we recover to sound or look the same as the original. As we pointed out at the end of the last section, the computer files you customarily use to store images and sounds contain far fewer bytes than the corresponding bitmap and wave files we dealt with in the experiments. How many bits do we really need to encode a text message, a picture, an audio recording?
Here we will look at various methods for *compressing* data.

The Morse Telegraph Code—An Early Data-Compression Scheme

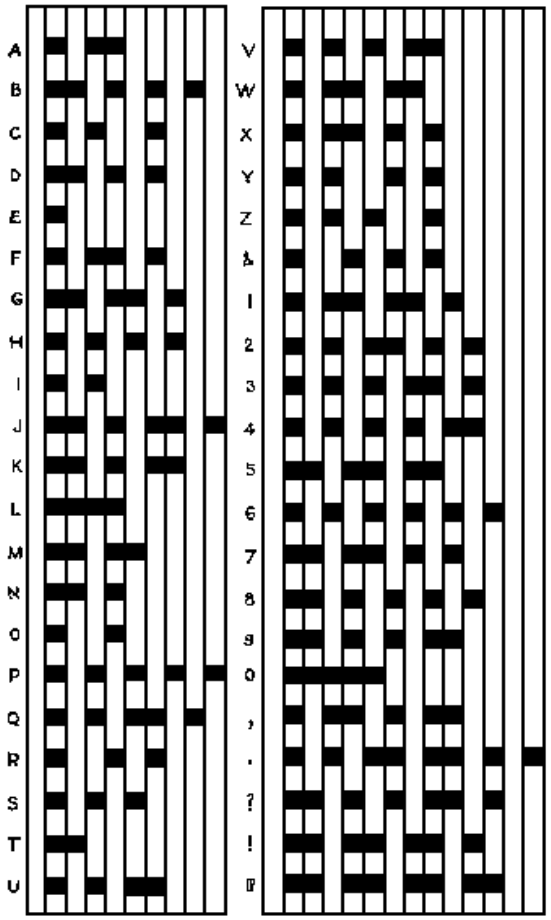
Before the 1840's, if you wanted to send a message to an acquaintance in another city, someone—or a team of people---had to physically carry the message to the recipient. Usually this meant travel on foot or by a horse-drawn vehicle. (The first steam-driven trains did not appear in the U.S. until the 1830's, and the transcontinental railroad was not completed until 1869.) The introduction of the electric telegraph made it possible for messages to be sent almost instantaneously to a distant destination. This was a real revolution in the way people communicate with one another, comparable in its impact to that brought about by the introduction of the Internet.

You may have performed the experiment in a school science class of winding a coil of wire around a steel nail and connecting the wire to a battery. The nail becomes magnetized for as long as the current is flowing. This is the principle of the telegraph: Someone operating a switch in one locale can power an electromagnet located at a very great distance. There are a number of technical problems to overcome before this can be turned into a useful system for communication. Most of these are hardware problems. But there is also the *software problem* of how to encode messages so they can be sent over a wire. (Not to mention the business and infrastructure problem of finding someone to pay for stringing those many miles of wire.)

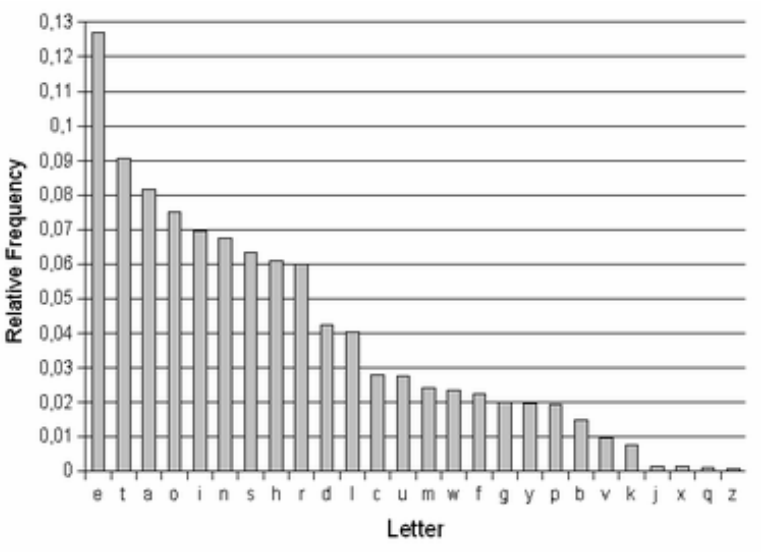


Old telegraph equipment---the key on the right was used to send the encoded messages, the sounder on the left to receive them.

Samuel F. B. Morse, who developed the first practical telegraph in the U.S., tried a number of encoding schemes before settling on the following solution: The sender would open and close a simple switch, sending a series of short and long pulses ("dots" and "dashes") which would be heard as a series of clicks at the receiving station. Each letter of the alphabet was thus encoded as a sequence of dots and dashes, with a pause to separate letters, and a longer pause to separate words. Below is a table of the encodings Morse used.



This system is not completely arbitrary: the letters whose encodings have the shortest durations (E,T,A, O, I, N) are also those that occur most frequently in English text. (L, which is encoded by a long dash, is as brief as E, T, A, O, I, N, even though it does not occur as frequently nearly as frequently as the others.) Requiring less time to send the most



frequently-occurring letters reduces the overall transmission time. Observe that if the message were just gibberish consisting of letters chosen at random, there would be no advantage to this encoding.

Redundancy

Only a small fraction of the possible sequences of 26 letters actually occur at all in English words. The patterns of letters are highly predictable: 'Q' is almost certain to be followed by 'U', a 'T' at the start of a word will never be followed by anything but a vowel or the letter 'H'. There are tens of thousands of different words in English, but a mere thousand of them account for more than three-quarters of the words that you typically read. Even at the level of individual letters, there is some predictability: as we saw above, a letter chosen at random is much more likely to be an 'E' than a 'V', 'K' or 'X'.

This *redundancy* helps guard against errors in communication, but it also suggests that the information contained in a text message can be encoded in a more compact form. That is the idea behind the transmission-time compression in Morse Code, and also the idea behind modern data compression schemes used in computers. We'll look at several methods for doing this.

Huffman Coding

This is a compression method that resembles the principle by which Morse code was constructed: Frequently occurring symbols are encoded by short sequences of bits, and rarer symbols by long sequences of bits.

Let's suppose that we have a long text consisting of the letters a,b,c and d. Since there are four symbols, we could encode each symbol by two bits, say

a:00
b:01
c:10
d:11

and thus encode a long message, like:

abbacaaddaaacbbbbacababababad…

as

000101001000001100000010010101010010000100010001 0011…

But suppose we know something about the relative distribution of these letters in typical messages: that a occurs much more frequently than b, and that each of these is much

more common than c or d. For instance, the letters might occur with the following frequencies.

a:50%
b: 30%
c: 12%
d: 8%

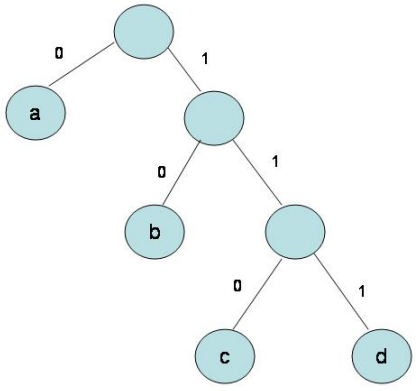
In that case, we could try to use a shorter encoding for ‘a’, a longer one for ‘b’, and longer still for ‘c’ and ‘d’:

a: 0
b: 10
c: 110
d:111

For instance, the sequence of letters in the example above would now be encoded as:

0101001100011111100011010101010011001001001001111…

Note that this coding system has a crucial *prefix property:* no one of the four sequences of bits is an initial segment of any other, so there is only one way to decipher an encoded message. When we read the 0 at the beginning, we know the message must have begun with ‘a’. When we next read the 1, we advance to the next bit, and now having seen 10 we know that the next symbol of the original message must be ‘b’. A code with the prefix property can be depicted as a tree, with the code letters at the leaves of the tree, the edges sloping down to the left labeled 0, and those sloping down to the right labeled 1.



In fact, the tree can be used as a diagram of the decoding algorithm: start at the root; each time you read a bit follow the appropriate edge of the tree; when you come to a leaf, write down the message symbol and go back to the root.

A message consisting of one hundred letters would contain, on the average, 50 a’s, 30 b’s, 12 c’s and 8 d’s, and thus be encoded in this system by a sequence of

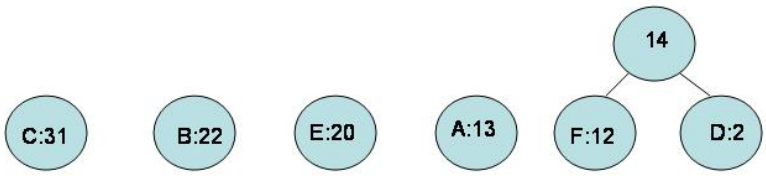
50x1+30x2+12x3+8x3=170 bits,

which is a good deal less than the 200 bits required by the original scheme.

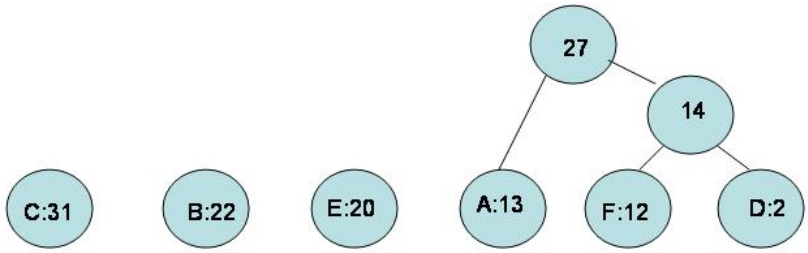
Huffman coding is a way to obtain such an encoding system for any table of letter frequencies. We start by listing the frequency of each symbol (the example below uses a six-letter alphabet):

A: 13% B: 22% C: 31 % D: 2% E: 20% F: 12%

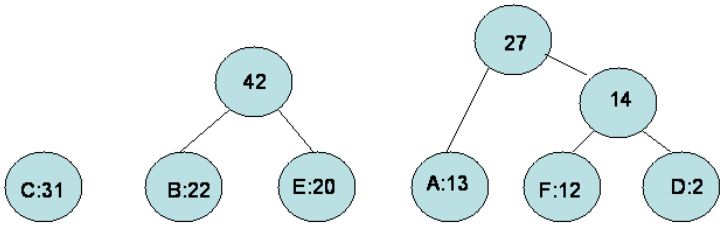
We then take the two least frequently occurring symbols, and group them, as shown below.



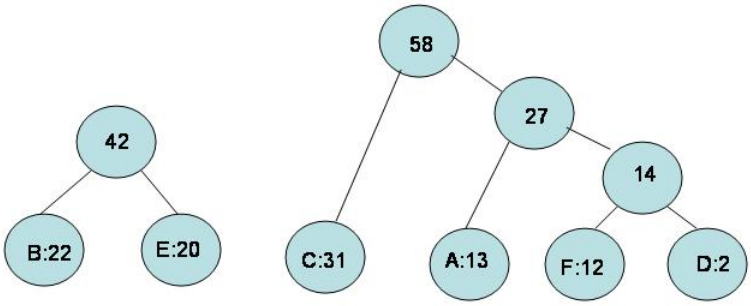
We treat this group as though it were a new symbol, and repeat the procedure:



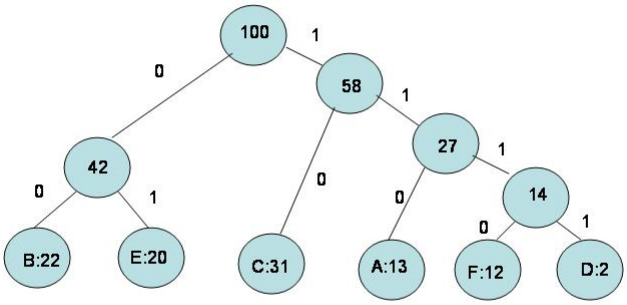
And again:



And again:



And finally:



The edges of the finished tree have been labeled 0 and 1, and this gives us our code:

A: 110
B: 00
C:10
D:1111
E:01
F:1110

In the example shown, a message of 100 letters would require on the average

13x3+22x2+31x2+2x4+2x20+4x12=241 bits

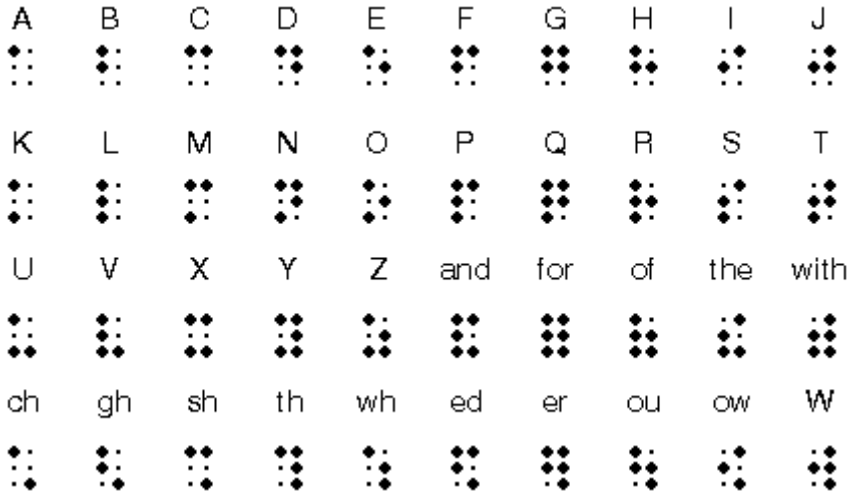
for its encoding. The straightforward alternative, which uses 3 bits for each symbol, would require 300 bits for the whole message.

Dictionary Methods and Lempel-Ziv Encoding

A simple dictionary scheme for compressing English

Before the telegraph code described earlier was invented, Morse experimented with a scheme in which every *word* was *encoded* by a number. Presumably messages would involve only commonly-occurring words, and the decoder would have access to a dictionary that gave the word corresponding to each received number. This unwieldy method was soon abandoned in favor of the alphabetical code, but it can be made the basis for sophisticated compression algorithms.

Both Morse Code and Huffman coding use a variable-length encoding of a fixed set of symbols. An alternative method is to use a fixed-length encoding of sequences of symbols. An old example of this method is the Braille alphabet, used by blind readers.

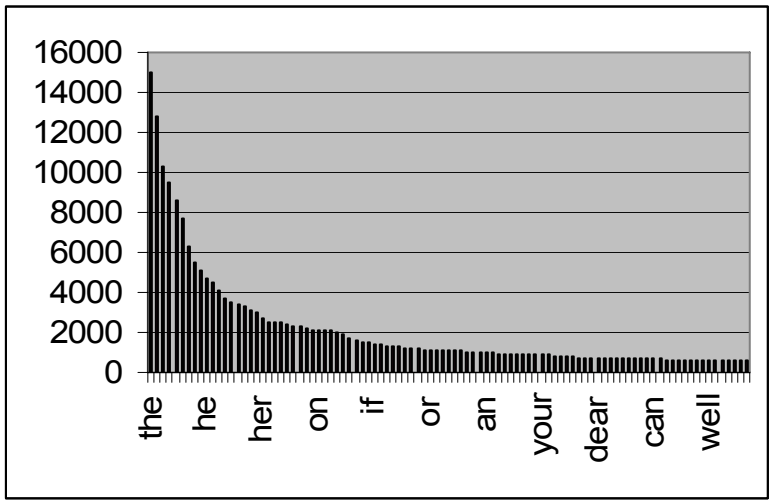


Each Braille symbol is given by 6 bits (raised dots in a 3 x 2 grid), which makes for 64 possible symbols. The chart above does not show the encodings of punctuation symbols or numbers, but it does illustrate the kind of compression method we are talking about for the encodings of letters. Commonly-occurring words and clusters of letters, like “the”, “of” and “sh” are each encoded by single symbols. If a symbol for a word or cluster is unavailable, then symbols for the individual letters are used. This achieves a modest degree of compression over the alternative of encoding each letter by a single symbol. (Books printed in Braille are quite bulky and expensive, so compression is important.)

The same idea can be used to compress texts written in ordinary English; the idea is to choose a fixed encoding for each of a number of commonly occurring words, and then to encode the rarer words letter by letter. To keep the discussion simpler, we’ll assume that the only punctuation is the spacing between words, and that there is no distinction

between upper-and-lower case letters. ITS AS IF ALL THE TEXT LOOKED LIKE THIS WE COULD MAKE PROVISIONS FOR PUNCTUATION AS WELL AS CAPITALIZED AND UNCAPITALIZED WORDS THIS WOULD ONLY CHANGE SOME OF THE DETAILS

I did my own little study of the distribution of words in English by tabulating all the words in a long novel. (*Bleak House,* by Charles Dickens). The novel contains 363,749 words and 1,866,452 characters (counting only the characters in words and the spaces between them). There are 15,210 distinct words used, and close to 6,000 of them occur only once in the book. The chart below shows the frequencies of the 100 commonest words.



The 2048 commonest words account for about 90% (325,233) of the words in the text and about 83% (1,544,844) of the characters. Since $2048=2^{11}$, we can associate a unique 11-bit string to each of these words, and since there are only 26 letters plus a space, we can associate a 5-bit string to each individual symbol. We need some way to tell the decoder whether to interpret the bits it is reading as encodings of words or of individual letters. We can do this by appending a single bit (1 for word encodings, 0 for letters) at the front of each of the encoding. Thus each dictionary word is encoded by a 12-bit string, and each individual letter and space by a 6-bit string.

What sort of compression does this achieve? A simple letter-by-letter encoding would require 5 bits per character, and thus use

1,866,452x5=9,332,260 bits

Our compression scheme would encode 325,233 words by 12-bit strings. This would replace 1544844 of the characters in the original text, leaving 1866425- 1544844=321581 individual characters to be encoded by 6-bit strings. The result is

325233x12+321581x6=5,832,282 bits,

about 62% of the original size. (This number, 62%, is called the *compression ratio.*)

Lempel-Ziv Encoding

The scheme we describe above uses a fixed dictionary, and is only useful for files of a certain kind. Here we'll describe a practical dictionary-based algorithm, one closely related to the algorithm used to make those .zip files that you use all the time on Windows. The idea is again to build a dictionary of commonly occurring patterns, but the dictionary is built on the fly while the original file is being encoded, and then reconstituted while the compressed file is decoded.

As new text is encountered, it is compared to text that has already been seen. The algorithm finds the longest match between the new text and what has already been read. In the example illustrated below, the text in blue, from the beginning, up until the initial 'R' of 'RESTING', has already been read. The new characters 'ESTING<space>' match the characters shown in red (including the space).

FOUR SCORE AND SEVEN YEARS AGO OUR FATHERS BROUGHT FORTH ON THIS CONTINENT A NEW NATION CONCEIVED IN LIBERTY AND DEDICATED TO THE PROPOSITION THAT ALL MEN ARE CREATED EQUAL NOW WE ARE ENGAGED IN A GREAT CIVIL WAR TESTING WHETHER THAT NATION OR ANY NATION SO CONCEIVED AND SO DEDICATED CAN LONG ENDURE WE ARE MET ON A GREAT BATTLEFIELD OF THAT WAR WE HAVE COME TO DEDICATE A PORTION OF THAT FIELD AS A FINAL RESTING PLACE FOR THOSE WHO HERE GAVE THEIR LIVES THAT THAT NATION MIGHT LIVE IT IS ALTOGETHER FITTING AND PROPER THAT WE SHOULD DO….

The encoding algorithm outputs the following information:

7 (the length of the match)
194 (how many characters back it had to search to find this match)
"P" (the character following the matched text)

The sequence of such triples constitutes the compressed file. In essence, the previously read text is being treated as a dictionary, and the numbers 194 and 7 tell where in the dictionary the match is to be found. It is a relatively simple matter for the decoder to reconstitute the original text from this sequence of triples. In practice it is necessary to limit how far back in the text the algorithm will search for a match, and how long a match it will report.

I experimented with this method on several files, limiting the search text to the 4095 characters most recently read, and limiting the size of the match to 15 characters. This meant that each triple could be encoded in 24 bits = 3 bytes (4 bits for the length, which is a number between 0 and 15; 12 bits for the offset, a number between 0 and 4095; and 8 bits for the offset, a single character). When I applied the method to the complete text of Lincoln's Gettysburg Address (not much longer than what you see above, but with all the

punctuation and lower- and upper-case letters intact), the algorithm produced 373 triples, which would occupy 3x373=1019 bytes, while the original file contained 1462 bytes. Thus the compression ratio is about 70%. It should be mentioned, however, that this algorithm does not work terribly well with short text files, because you usually need a lot of text to see long, common sequences of letters reappear. When I applied the algorithm to a longer file, with 84,000 bytes, it produced 17140 triples, so the compression ratio is (17140x3)/84000=61%. By the way, the zip algorithm, which is based on similar ideas, but works considerably better, compressed this same file to 44% of its original size. This is typical for English text.

The Lempel-Ziv algorithm can be applied to any sequence of bytes, not just to text files. In fact a very common application is to compress application software, which consists of very long sequences of computer machine instructions. We'll study these later, but all you need to know for now is that each instruction takes up just a few bytes, some instructions are much more common than others, and that the same sequences of instructions will often appear many times in the same file. All that redundancy translates into a good target for data compression. The zip algorithm typically compresses these files to about 50% of their original size.

Lossy Compression of Images and Sounds

The 264 by 320 grayscale image shown below was originally created as a file of 84480 (=264x320) bytes. When I tried to compress it using the zip utility in Windows, the resulting file contained 78700 bytes, 93% of the original size. Thus Zip compression did not find much redundancy in the original grayscale bitmap. But when I saved it in JPEG format (which in fact is what you see in this document) the file shrank to 19,724 bytes, only 23% of the original size.

You observe something similar with audio files. A music file stored on an iPod might use between 5% and 10% of the space (measured in bytes) that the same music would occupy on a CD.

How is so much compression achieved? There are many different schemes used for compressing images and audio. Both JPEG image compression and MP3 audio compression are based on the following ideas:

(a) The sound or image can be *transformed into a different representation* in which there is greater redundancy. For example, a brief segment of audio can be analyzed into different frequencies---typically it will be found that the signal is very strong at just a few frequencies, and very weak or absent altogether at others. Thus if the sound is encoded by giving its strength at different frequencies, rather than its strength at different times, there will be greater redundancy.

(b) It is not necessary to be able to recover all the original bytes from the compressed file; the only requirement is that the recovered information looks or sounds as good as the original. *Thus it is permissible to throw out information.* This is called *lossy* compression. (The kind we've seen up until now, in which the original file is recovered verbatim, is called *lossless* compression.) For instance, the ear is much more sensitive to certain frequencies than to others. Thus after analyzing the original signal into component frequencies, we can use fewer bits to encode the signal strength at the less sensitive frequencies, which will further increase the redundancy.

A Simple Lossy Image-Compression Scheme

We will illustrate these ideas with the image shown above. The algorithm we use is a highly simplified version of some of the ideas used in JPEG compression.

Let us imagine first that we have two consecutive pixels in a single row, with values a and b.

a	b

If we save the sum and difference of these two values

a+b	a-b

then we can easily recover the original values a and b by again taking the sum and difference, and then dividing by 2. What is the point of doing this? In images like the one above, adjacent pixels typically have very close values, so the differences a-b tend to be small, and thus drawn from a smaller set of possible values. Merely by saving the sums and differences rather than the original pixel values, the redundancy is increased, but no information is lost from the original file.

Moreover, very small differences are barely perceptible, so that if a-b is small enough, we can replace it by 0. The result is a file in which a large number of values are zero, and many of the rest are quite small integers. The end product has a lot more redundancy, but the image quality deteriorates only slightly.

Fancier versions of this method use successive sums and differences on blocks of 4, 8, 16,…adjacent values. For example, with a block of 8 adjacent values

A	B	C	D	E	F	G	H

we begin by taking sums and differences of adjacent values:

A+B	A-B	C+D	C-D	E+F	E-F	G+H	G-H

Then we take the sum and difference of the first and third entries, and the fifth and seventh:

A+B+C+D	A-B	(A+B)-(C+D)	C-D	E+F+G+H	E-F	(E+F)-(G+H)	G-H

Then the sum and difference of the first and fifth entries:

A+B+C+D+E+F+G+H	A-B	(A+B)-(C+D)	C-D	(A+B+C+D)-(E+F+G+H)	E-F	(E+F)-(G+H)	G-H

The values in this table give information about the image at different *scales*: The first entry gives the brightness of the entire image, the fifth entry tells how the brightness changes from the left half to the right half, the third and seventh entries tell how the brightness changes from the first to the second quarter, and from the third to the fourth quarter. The other entries tell how it changes between adjacent pixels. Once again, it is easy to recover the original values from these transformed values. If we replace small values in the transformed table by zero before we do the recovery operation, the image will not change too much, and the redundancy will be higher.

How do we apply this technique to real images? I took the image shown at the beginning of this section, and divided it into 8x8 blocks. Within each block I applied the transform operation described above first to each row, and then to each column. I then set the 74000 smallest of the 84480 transformed values to zero.

Increasing the redundancy in this fashion made it possible to compress the collection of transformed values to one-third of the original file size. After the recovery operation is performed, the resulting image is this:

88% of transformed values set to 0, file compressed to 33% of the original size

The deterioration in quality is noticeable, it is not so bad, and the compression is much more significant than what we obtained by simply zipping the original file. If fewer transformed values are set to 0, the compression is not quite as good, but the quality is better:

79% of original transformed values set to 0, file compressed to 46% of original size.

(When I view the pictures at this scale, I have trouble distinguishing this image from the original one. If you blow them up to a larger scale you can see the differences pretty clearly.)

More sophisticated transforms lead to better results, of course, but the same idea is present. First transform, then filter out small transform values, then compress using a lossless compression method like Huffman coding or zip.