Error Correction and Compression – Part 2

To store information on a computer, it is often useful to compress it, so that it does not take up too much memory. There are typically two types of compression schemes: lossless, or lossy. We shall discuss below some text compression schemes that are **lossless**, meaning that you can recover the original information completely and without any changes. In other applications, especially in image analysis, we are content with **lossy** compression, where you cannot recover the original exactly from the compressed version, but where the reconstruction is "close enough" to the original (for instance, where the difference is imperceptible to our eye).

The big sqze: Lossless Compression of Texts

In the error correction and/or error detection schemes we have seen, some redundancy is introduced to make it possible to detect/correct errors. For instance, when the last digit of a 10-digit number is a check digit, this means that all the numbers that have the same first nine digits but a different tenth digit are not allowed—only 10% of all possible ten-digit numbers are acceptable. This is somewhat similar to redundancy in human language: far from all possible combinations of five letters (from the 26 letters in the alphabet) give rise to real words.

This redundancy in language makes it possible to correct (mentally) many typos that we encounter—that is the error correcting side of redundancy.

There is another way in which redundancy in language can be exploited: it can be used to compress the content of a message. This is seen very markedly in advertisements. It takes much less space to print "lrg BR" than "large bedroom," yet we understand it just as well. (Of course, if you make an error in the compressed version, then it will be harder to understand—in "lrg DR" we do not as easily detect the typo that is clear in "large dedroom.")

Since many files on a computer consist of text (either English or a programming language), we can try to save memory space by compressing them. There are many compression schemes around. For text, we shall illustrate the two main ideas here.

Let's start by transposing the text into binary symbols—that is what is done in the computer anyway. Since we have 26 letters (let us not worry about upper case letters or punctuation right now), we need five bits per letter (four bits would give only 16 possibilities—too little; five bits gives 32, which is a little too much, but we can throw in a few punctuation marks

to get a round 32 symbols.) **This group of words** would take $19 \times 5 = 95$ bits (because we have 19 symbols, if we include the spaces). But in a standard English text some letters are much more frequent than others (compare e and z, for instance). Samuel Morse exploited this idea in 1844 to create a code (now universally known as Morse code) to be used with the then brand new telegraph. In his code the two most frequent letters E and T are represented by $\cdot$ (dot) and – (dash) respectively. Two less frequent letters Z and Q are represented by – – $\cdot$ $\cdot$ (dash dash dot dot) and – – $\cdot$ – (dash dash dot dash) respectively. He made frequent letters shorter and infrequent letters longer, thus saving time for transmitting a telegram. We can not use Morse code literally in binary numbers because Morse really has **three** building blocks for his code: dots, dashes and spaces (to separate the different letters that are being sent. For instance, SOS becomes $\cdot\cdot\cdot$ – – – $\cdot\cdot\cdot$: the spaces are necessary to announce the end of one letter and the start of the next letter.) We have only zeros and ones. This makes things a bit more complicated. Consider the following examples:

1. Suppose we use a code that represents the letter O as 1 and the letter M as 11. Now we get sent the string 11111. How we can decode this? We could read every single 1 as O; we could also choose to view some pairs 11 as M. In fact any of the following are possible: OOOOO, MOOO, OMOO, OOMO, OOOM, MMO, MOM, OMM, a total of 8 different possibilities.

2. Suppose that in some other code, we represent A as 01 and B as 010 . We have to decode some information that is encoded as 0100111100. What is the first letter? A or B?

In order to avoid this problem, we have to represent letters in such a way that we automatically know when to break for a new letter. This is done by introducing the idea of a *prefix*. A prefix of a codeword is a part of that codeword that we find at the start of the codeword. For instance, 01 is a prefix of 010 and 1 is a prefix of 11; 011 is a prefix of 01101, and so is 0110 .

Suppose that we have a code in which no codeword can be viewed as a prefix for another codeword. Then we can always decode a binary string: We start at the left till a part of our string matches a codeword. Since it cannot be the beginning of another codeword, there is no ambiguity (as in the earlier examples we saw), and we have recovered our first letter. Let's illustrate this with a simple example, using a few letters from the Huffman code, which we will discuss in greater detail below. In this code, no codeword is a prefix of another codeword. Take for example the five letters A, B, N, T, Y, encoded in the Huffman code respectively as A=011, B=00001, N=1110, T=110, Y=00000. Try now to decode 110011111000000011 . There is only one way in which it can be done!

Let's look at the binary tree for the **Huffman code** (Figure 1). We start from the root. It splits into two branches; we label the left branch with a 1 and the right branch with a 0. Each of these branches splits again; to label these "next generation" branches we take the label of the mother branch and attach a 1 for the left "child" branch and a 0 for the right. We keep doing this, following the tree. When a branch stops without splitting further, we say that we have arrived at a leaf. To each leaf a letter is assigned. (See Figure 1.) This letter is encoded by the path taking at the different branchings that led to this leaf. For

instance, to get to the leaf with letter A we first take the right branch up (0), then left (1), then left again (1) and then we have already reached the leaf; so A is encoded as 011.

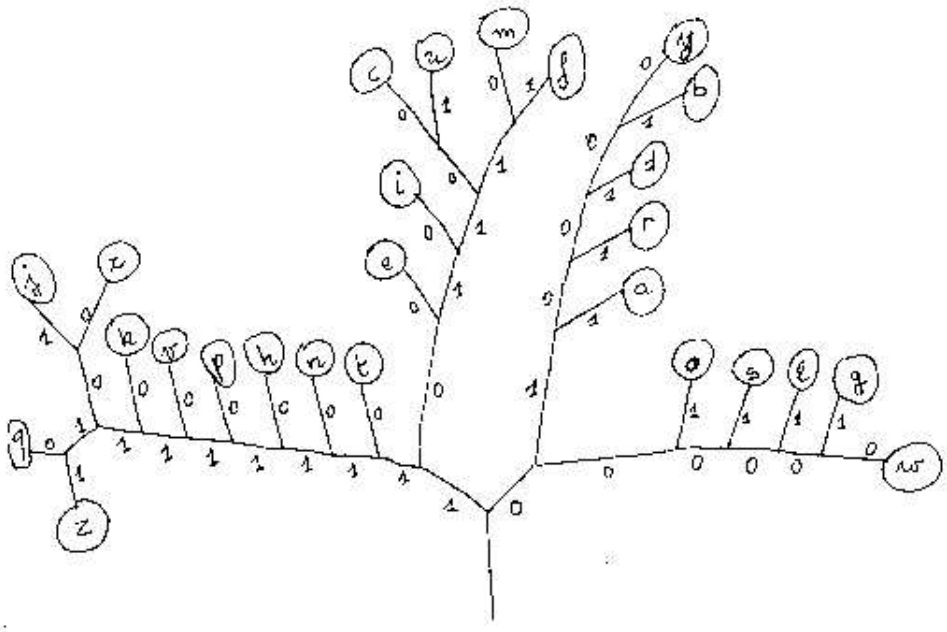

Figure 1: Tree for Huffman coding of the English alphabet

Take now the two letters F=101111 and C=101100. What is the largest string that is a prefix for both? We can see that it is 1011 — indicating the common path followed on the way from the root to the leaves F and C until the point where the two paths separate. This tree structure explains why in this code no codeword is a prefix to another codeword: codewords represent leaves, and from a leaf we cannot go to any other leaf, since the branching process stopped. So this code encodes uniquely; decoding will be unambiguous.

This argument would work for any binary tree, as long as we attach the letters to leaves only. What is then special about the Huffman tree in Figure 1? It turns out that this tree is one of the best possible binary trees to encode standard English texts. The following paragraph explains exactly how this tree was constructed.

First, you need to know how often different letters are used in standard English texts. For this, you simply take a great deal of English texts, and you check how many times, for instance the letter A occurs, and you compute the percentage that this represents. You have to do this for all the letters of the alphabet. This gives you the *frequencies* of the letters.

(See the Table below.)

LETTER	STANDARD FREQUENCY
a	.0761
b	.0154
c	.0311
d	.0395
e	.1262
f	.0234
g	.0195
h	.0551
i	.0734
j	.0015
k	.0065
l	.0411
m	.0254
n	.0711
o	.0765
p	.0203
q	.0010
r	.0615
s	.0650
t	.0933
u	.0272
v	.0099
w	.0189
x	.0019
y	.0172
z	.0009

Frequencies of Occurence of Individual Letters in English Text

The two least frequently encountered letters are Z (about 9 per 10,000, or with a frequency of .0009) and Q (about 1 per 1000 – frequency .001). We start drawing our tree from these two letters down: they will be neighboring leaves, coming from the same branching. (It won't matter which one we label by 0 or by 1; both leaves will have labels of the same length.) Now forget about the distinction between Z or Q - think of all the texts as written on a typewriter that had only 25 possible keys, with the same symbol V_1 (standing here for "Z or Q") replacing all Zs and all Qs. Then this new letter V_1 occurs with frequency .0019=.0009 + .001 (the sum of the frequencies of Z and Q). For this new alphabet of 25 letters, we can proceed as before: take the least frequent letters J (.0015) and X (.0019) (you can take V_1 instead of X since they have the same frequency .0019 – it does not matter – we would get another tree that is as optimal as the tree in Figure 1), draw the branches down from them to the common vertex V_2, which can again be thought of as standing for a new letter

"J-or-X" that would be assigned the frequency .0034 (=.0015+.0019), and replace the two letters J and X by V_2, reducing our alphabet to 24 letters. Go on: in the reduced table of 24 letters we can again pick the least frequent letters, now V_1 and V_2, and proceed as before. In this way we draw the tree from the leaves down; the most frequently occurring letters will only start coming into play after many reductions, and they will end up with much shorter labels than Z or X.

The end result is that we use fewer bits for frequent letters than for less frequent letters, resulting in compression. Taking into account the frequencies of the letters in standard English, this means that in a long text, we will use about $3.9 \times n$ bits for n symbols (lower case letters and spaces), which is a compression over the $5 \times n$ bits we would have needed without this scheme! Huffman coding was used in most compression schemes until recently, and is still very widely used. You can of course use it for other things than text as well—the idea is always that you determine a statistics of the different symbols, and then encode the most frequent symbols with fewer bits than the less frequent ones.

Another compression idea, especially good for dictionaries, is called the *difference algorithm*. Looking at a dictionary you can guess that it is probably faster to tell how a word differs from the previous one than to describe the word itself.

Let us illustrate this by compressing a wordlist:

a, aardvark, aardwolf, aaron, aaronic, ab, aba, abaca, abaci, aback, abacus, ...

The following would be a way to compress this (the number indicates how many characters from the previous word we have to take over first):

a	0a
aardvark	1ardvark
aardwolf	4wolf
aaron	3on
aaronic	5ic
ab	1b
aba	2a
abaca	3ca
abaci	4i
aback	4k
abacus	4us
⋮	⋮

Clearly the second list takes less space!

This works very well on alphabetically ordered wordlists, but not all texts are so obliging! Let's try the same idea in a slightly different guise.

In recent years, the Lempel-Ziv algorithm, which incorporates a similar idea, in a subtler form, has taken over as the "compression champion," mainly because it is much easier to implement. The command "compress" on Unix machines uses LZ; so do the compression programs for PCs that are now on the market (like Stacker).

You observe that in many texts, there will be words that recur a lot. You can then say, when encoding the text, "here is a word that I saw before, it was the fourth word in this encoding, so just look there," and encode by giving simply an agreed way of saying "look back at the fourth word."

Take as a sentence,

The grain in Spain grows mainly in the plain.

We are going to parse this, by putting a comma after every string we haven't seen before in the sentence. As long as the string looks identical to something we placed between commas before, we are not allowed to place a comma. So:

,t,h,e, ,g,r,a,i,n, i,n ,s,p,ai,n g,ro,w,s ,m,ain,l,y, in, t,he, p,la,in.

As we go on, the strings between commas become longer—the algorithm is "learning" words. At the same time as we do this, we number the strings. The first (empty) string is given the number 0, the next one (t) is number 1, and so on; for instance, the string with number 10 is ((space)i), ... Then every string is always the combination of an earlier string plus one new symbol. (For instance, "(space)i" = space + i = string 4 + i.) So the sentence can be written down as:

00t0h0e0 0g0r0a0i0n4i9 0s0p7i11g6o0w12 0m14n...

At first, this doesn't look like you save space, but in the long run it definitely does, because the number of the prefix (the string you saw earlier) is much shorter than the prefix itself.

You'll try this out in the lab.

Reconstructing from the compressed version is easy too, because you always have to use prefixes that you reconstructed earlier, so you know them already!

In Lempel-Ziv compression you build a dictionary as you encode, and number the items in your dictionary. The coded version uses the items in the dictionary that were built before,

and refers to them by their number. Let us try another example, with a slightly artificial sentence:

> I thought a thought but the thought I thought was not the thought I thought I thought.

Put a comma after every string that you hadn't seen before:

> ,I, ,t,h,o,u,g,ht, a, t,ho,ug,ht ,b,ut, th,e, tho,ugh,t ,I ,th,ou,gh,t w,a,s, n,ot, the,
> ↑ "empty" string
>
> thou,ght, I, thoug,ht I, though,t.

If you do it correctly, then you end up with 37 commas. Now number all those strings:

0	"empty"	9	(space)a	18	27	36
1	I	10	(space)t	19	28	37
2	(space)	11	ho	20	29	
3	t	12	ug	21	30	
4	h	13	ht(space)	22	31	
5	o	14		23	32	
6	u	15		24	33	
7	g	16		25	34	
8	ht	17		26	35	

Every string can be identified as a "prefix" (everything but the last letter, and this must have been seen before!) and its last letter. For instance, string 8 = ht can be written as ht = 4t (because string 4 is h); similarly string 13 = ht(space) can be written as 8(space).

So we get for the encoded version (obtained by stringing all the re-written strings together):

> 0I0_0t0h0o0u0g4t2a2t4o6g8_ ...

After completing this, we get 83 characters, whereas the original sentence has 86 (including spaces). Hardly an impressive compression .... But if we look at the second half only then the last 41 characters correspond to only 33 in the encoded version; at the start of the message there is in fact an "expansion", but this happens only at the start. For long messages, this scheme will indeed compress.

In practice, one uses smarter schemes, that may be harder to implement, and that do not give bigger gains in the long run, but that start paying off sooner than the naive scheme

above. For instance, you can look back and put a comma only when you never saw those letters in that order before, regardless of whether they were separated by commas:

> ,I, ,t,h,o,u,g,ht, a, thought b,ut, the, thought I, thought w,a,s, n,ot, the thought I thought I, thought.

But let's stick to the simple and naive scheme explained earlier. In the lab, you will work on binary sequences as well as with sentences in our 26-letter alphabet. We then have a problem that we didn't face in our earlier example: we will use the same symbols (binary numbers) for the numbering of the strings as for the extra digits. In order to know when to read the bits as part of the numbering or as extra digits, we need to be careful in setting up the scheme. Let's do an example: First we need to parse—that is, to put in our commas:

> ,1,0,11,10,100,10111010000110110100011101011110

(the first few commas have been marked; please insert the others)

If you do it correctly, you'll end up with 16 strings (including the empty string):

string	number of string in decimal	number of string in binary
empty	0	0
1	1	1
0	2	10
11	3	11
10	4	100
100	5	101
	6	
	7	
	8	
	9	
	10	
	11	
	12	
	13	
	14	
	15	

(please complete this table)

In this table, the largest string **number** in binary has four bits; we shall therefore write **all** of them with four bits by adding zeros in front. For instance, the third string, 11, gets number 0011. To encode the string, we write it every time as prefix number plus last bit, just as before. For the first strings we have:

$$
\begin{array}{lll}
\text{string } 1 = 1 & = \text{empty string} + 1 & \\
& = \text{string } 0 + 1 & \rightarrow 00001 \\
\text{string } 2 = 0 & = \text{empty string} + 0 & \\
& = \text{string } 0 + 0 & \rightarrow 00000 \\
\text{string } 3 = 11 & = \text{string } 1 + 1 & \rightarrow 00011
\end{array}
$$

and so on:

string	string number	prefix number (in binary)	last bit	coded version
empty	0	—	—	—
1	1	0000	1	00001
0	2	0000	0	00000
11	3	0001	1	00011
10	4	0001	0	00010
100	5	0100	0	01000
	6			
	7			
	8			
	9			
	10			
	11			
	12			
	13			
	14			
	15			

So the final coded version is:

00001 00000 00011 00010 01000

(please complete)

In order to decode this, the only thing you need to know is that the coded string length (which is constant because all the prefix numbers have the same length!) is five bits for each piece. You can then unfold things one by one, just like before:

$$
\begin{array}{lll}
00001 & \rightarrow 1 & \text{string } 1 \\
00000 & \rightarrow 0 & \text{string } 2 \\
00011 & \rightarrow 11 & \text{string } 3 = \text{string } 1 + 1 \\
& \vdots &
\end{array}
$$

Of course our example string, at 37 bits originally, and 80 bits after the LZ algorithm, can hardly be called "compressed"! This is again because the algorithm is so naive. In the long run (for instance, with binary files that are binary-ASCII representations of a full book, like you will do in the lab) you do really get compression; in order to be effective for shorter files, smarter versions of LZ need to be used.

Remarks: 1. The full "alphabet" available from a keyboard includes not only lower and uppercase letters and numbers, but also many punctuation marks and special symbols, some of which you can get only by punching two keys simultaneously. The total number of symbols available is often as high as 128. One can design an LZ compression scheme in whicyh, like in our binary scheme, the same symbols are used for the "numbering" as for the "tails", but that uses all 128 symbols available (instead of only the two digits 0 and 1). This is what most LZ-compression algorithms do; for the computer's internal use, each one of these 128 symbols is of course transcribed into strings of 8 bits each.

2. In Huffman coding, "fixed length" input (for instance, the letters in the alphabet) gave "variable length" coded output. In Lempel-Ziv, "variable length" input (the strings between commas) gives "fixed length" coded output. In this sense, these are complementary approaches. "Stacker" programs on PCs, or the "compress" command on Unix machines, all use a (smart) version of LZ.