20191010

Huffman Coding

The challenge: take a document consisting of a sequence of ASCII characters (such as the text on this page) and reduce the number of bits needed to store the information – while maintaining full recoverability. This is a vital problem in communications: compressing a document reduces the time required to transmit it.

More generally, we can consider a string S of characters based on any specific alphabet A, in which the characters in the string are represented by sequences of bits. Again, the challenge is to represent the information in S as compactly as possible without losing any of it.

We say that a "code" is an assignment of particular bit sequences to the characters in A, and an "encoding" of S is the process of representing S by the concatenated bit sequences for the characters in S.

Thus if S = "catattaca" and the code assigns "101" to "c", "000" to "a" and "011" to "t" then S is encoded as "101000011000011011000101000". Note that this encoding is certainly not optimal. Since there are only three different characters we can use 2-bit sequences without any ambiguity. So we could use "10" for "c", "00" for "a" and "01" for "t" (or any other trio of 2-bit sequences) and the encoding would be "100001000101001000"

Encoding is a problem that humans have been dealing with for millennia. The ancient Sumerians developed cuneiform writing as a way of recording detailed information with triangular marks on clay tablets – not quite bits, but a step in the right direction. The original Hebrew alphabet contained only consonants; vowels were not recorded, which reduced the amount of writing required. (I'm not claiming this was the reason behind not recording vowels, but it was certainly a side-effect.) In more recent times, secretaries were trained in various forms of "shorthand" for making accurate real-time transcriptions of meetings.

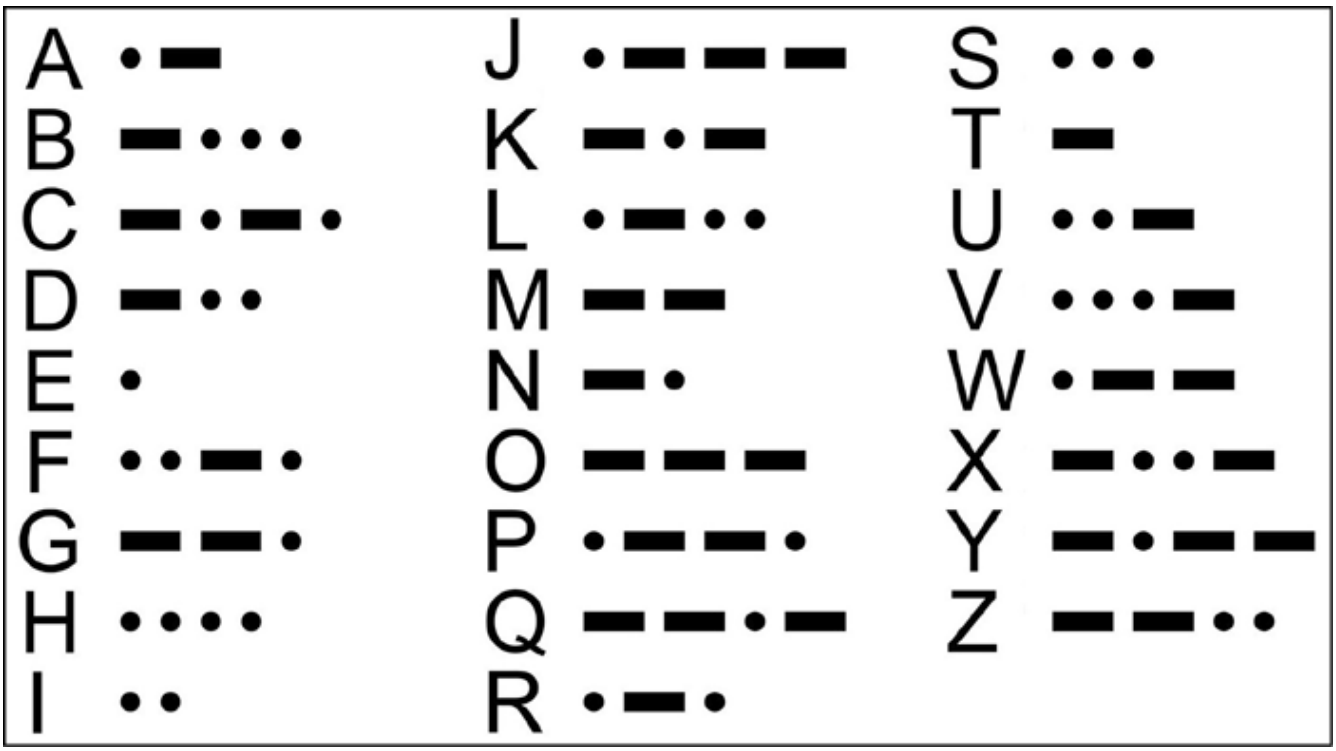
In 1948 Claude Shannon (an eccentric genius who invented – among other things – a motorized pogo stick) initiated the study of information theory with a paper titled "A Mathematical Theory of Communication". One of the first goals was to explore the limits of data compression. In 1952 an MIT student named David Huffman published a paper titled "A Method for the Construction of Minimum-Redundancy Codes" in which he gave an algorithm for creating **optimal** solutions for a particular class of coding methods.

Huffman Coding was (and is) such a powerful method for data compression that if you dig down into modern data compression formats such as zip, jpg and mp3 you will find elements of Huffman Coding behind them.

In ASCII, Unicode, (and for the historians among us, the archaic EBCDIC) each bit sequence assigned to a character has exactly the same length – for example all ASCII code sequences are 8 bits long. For example the ASCII code for "A" is "01000001". If we are using fixed-length bit sequences for characters, all codes are equivalent: if we are using bit sequences of length k and our string S has length n, then it will take exactly k*n bits to encode S.

But what if we use different-length bit sequences? If we assign short bit sequences to some characters and longer bit sequences to other characters, can we get an overall reduction in the encoding?

Most people have heard of Morse Code, even though relatively few people use it any more. (An interesting side note: the Titanic was ***not*** the first ship to use "SOS" as a distress call – the first known use was by the SS Slavonia, approximately three years before the Titanic went down.) Here's what the code looks like:



You can see that Morse Code uses the idea just mentioned, and it does so in a sensible way. Letters that are very common are assigned short codes (E and T have one-symbol codes, A, I, M and N get two-symbol codes) and longer codes are used for less common letters. In retrospect some of the choices are surprising. For example H is a much more common letter than M, and yet it has a longer code-sequence.

But there's a problem with this. If you were receiving a message in Morse Code and it started dot-dot-dot-dot, you could be excused some confusion. Did you just receive "EEEE" or "II" or "H" or "ES" or "SE" etc? Morse solved this by introducing a third "symbol" : a pause. Thus "ES" would be transmitted as "dot-pause-dot-dot-dot" and "SE" would be transmitted as "dot-dot-dot-pause-dot". Morse used longer pauses to mark the end of words.

Unfortunately this technique is not available to us, since we are limited to just 0 and 1 for our bits – there is no third symbol. Huffman coding takes a different approach to avoiding ambiguity.

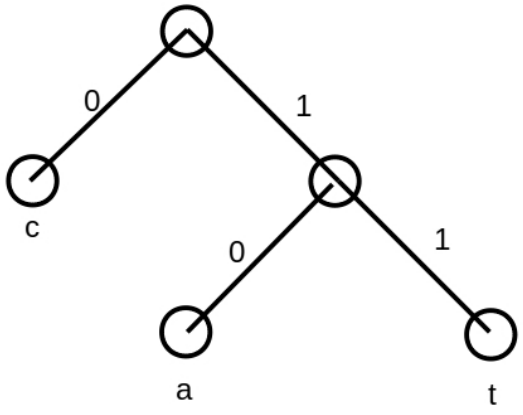
The Prefix Property: We say that a code satisfies the Prefix Property if none of the assigned bit sequences occurs as a prefix of any other assigned bit sequence. We call such a code a prefix code.

(Morse Code fails on this because – for example – the code for "I" is a prefix of the code for "H".)

When a code satisfies the prefix property we can decode without any ambiguity at all. As soon as we identify a bit sequence that has been assigned to a character, we know that is the correct character and we can start to decode the next character.

Let's look at S = "catattaca" again. Suppose we use the bit sequence "0" for "c", "10" for "a" and "11" for "t". This code satisfies the prefix property. The encoding of S is "0101110111110010".

We can visualize the code as a binary tree:



and this demonstrates an important idea: every prefix code can be represented as a binary tree in which the leaves correspond to the characters in the alphabet.

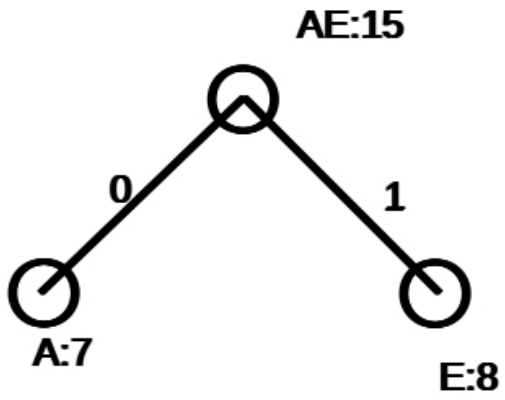
Decoding becomes a matter of starting at the root of the tree and following the branches dictated by the bits as we see them in the encoded string. When we reach a leaf we record the decoded letter and return to the top of the tree to continue.

As an exercise, decode at least the first few characters of the encoded string given above.

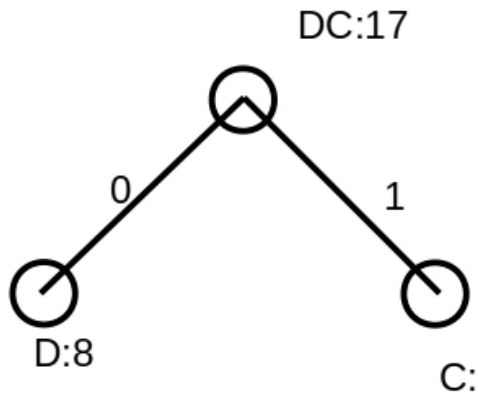
Huffman's brilliant idea is actually very simple: build the code by building the binary tree from the bottom up. Start with the two least-frequent letters, give them a shared parent (which assigns a "0" to the bit sequence for one of them and "1" to the other), and then treat that parent as a single character that combines the frequencies of the two characters just combined. Repeating this process until all the characters have been added to the tree gives the Huffman Code.

Example. Suppose S contains the characters A, B, C, D, E, F with frequencies 7, 16, 9, 8, 8, 11 respectively. The tree would be constructed in stages, as shown here:

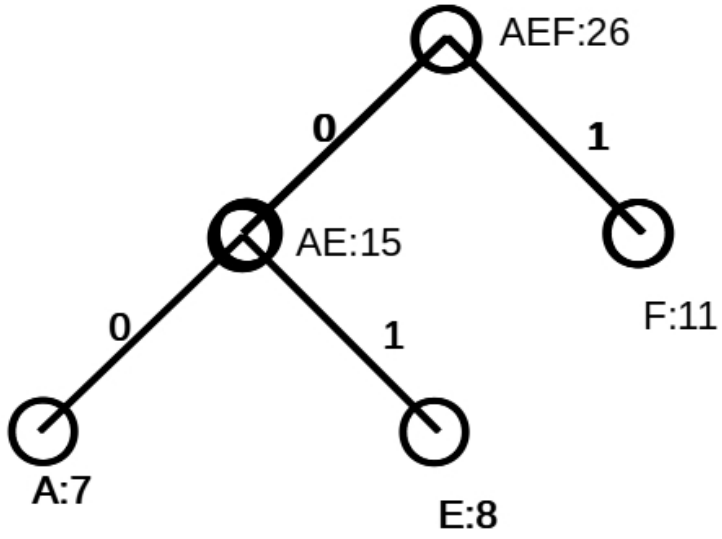
B: 16
F: 11
C: 9
D: 8
E: 8
A: 7

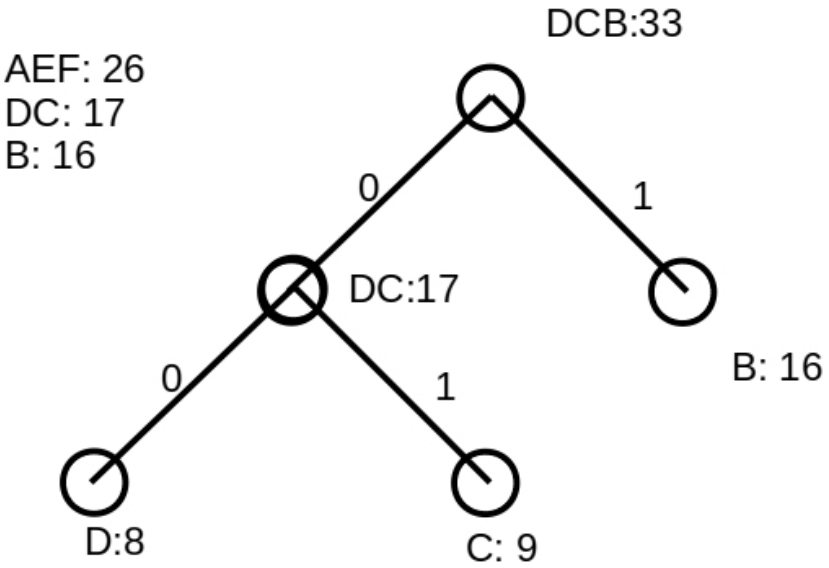


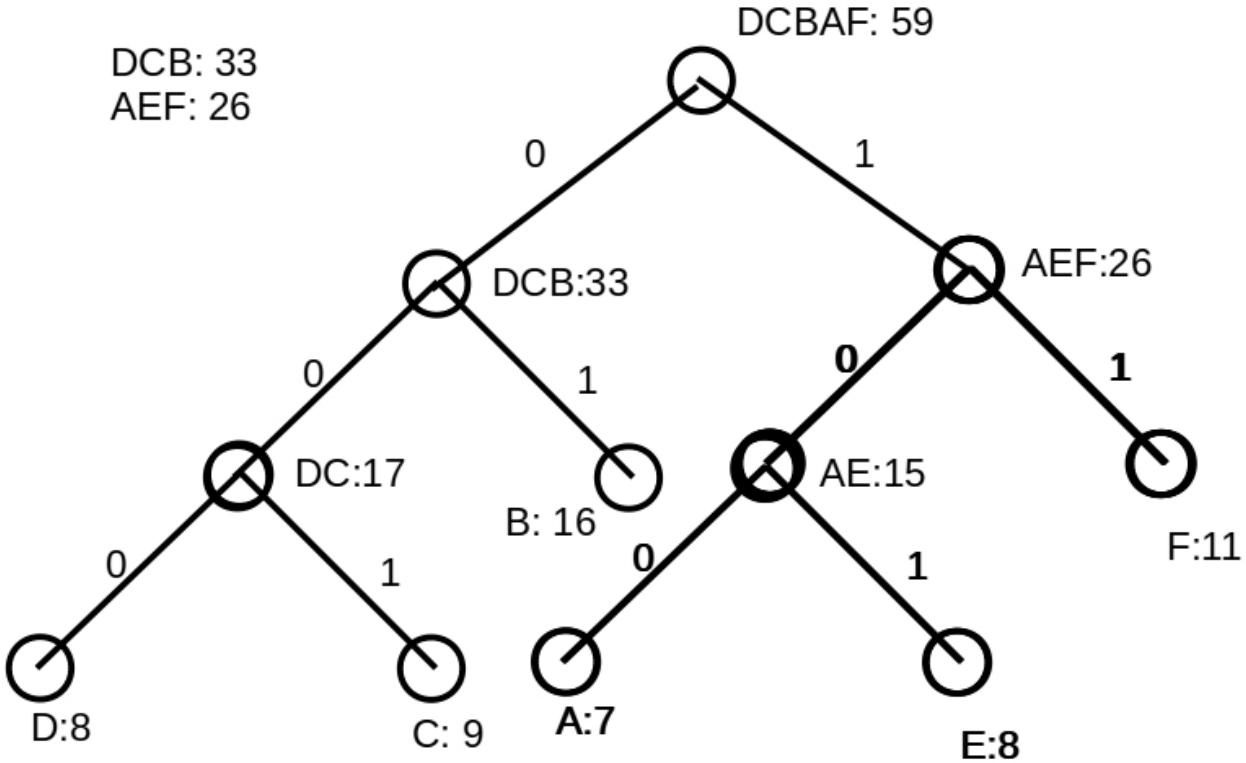
B: 16
AE: 15
F: 11
C: 9
D: 8



DC: 17
B: 16
AE: 15
F: 11




DCB:33
AEF: 26
DC: 17
B: 16
0
1
DC:17
B: 16
0
1
D:8
C: 9


DCBAF: 59
DCB: 33
AEF: 26
0
1
DCB:33
AEF:26
0
1
0
1
DC:17
B: 16
AE:15
F:11
0
1
0
1
D:8
C: 9
A:7
E:8

and now we extract the bit sequences from the top down:

A: 100

B: 01

C: 001

D: 000

E: 101

F: 11

With these bit sequences and the known frequencies of the characters in S, we can compute the total length of the encoded string:

7 A's with 3 bits each : 21 bits

16 B's with 2 bits each : 32 bits

9 C's with 3 bits each : 27 bits

8 D's with 3 bits each : 24 bits

8 E's with 3 bits each : 24 bits

11 F's with 2 bits each : 22 bits

for a total of 150 bits. We can compare this to the ASCII encoding – each of the 59 characters would require 8 bits, for a total of 472 bits.

It's reasonable to observe that with only 6 characters to encode, the full 8-bit allocation of the ASCII code is excessive. If we wanted to create a fixed-length code for just 6 characters, we really only need 3-bit sequences. For example we could use

A: 000

B: 001

C: 010

D: 011

E: 100

F: 101

This code would use 59*3 = 177 bits. The Huffman's Code compression – reducing the number of bits to just 150 – is still better. What Huffman proved was that his coding scheme is optimal among all prefix code. In other words, there is no prefix code that uses a smaller total number of bits than the Huffman Code.

The proof is a bit messy but not too hard. In structure it is very similar to the other Greedy proofs we have seen. It starts by showing that in any optimal solution, the two characters with the lowest frequencies should be at the lowest level of the binary tree that represents the code. Since this is where the Huffman algorithm puts them, we can see that the algorithm's first action is correct. This reduces the problem to a smaller problem (with those two characters combined into a single meta-character). We make an inductive assumption that the algorithm finds an optimal solution on the reduced set. Then we show we can patch the first step together with the optimal solution on the reduced set to get an optimal solution, which is exactly what the algorithm does.