How Short Can a Message Get?

Imagine you're sending telegrams in 1844, each symbol ('·' or '-') is expensive, so you want to minimise your expected length per letter, and since a telegraph only has two symbols, you can't give each of 26 letters a unique one. The simplest solution is if we assign each letter a 5 bit identification string e.g. 'a' gets (·····) 'b' gets (····–) etc. However, we would be wasting a lot of space, since common letters would be using as many symbols as rare ones. Samuel Morse noticed something: in English, E appears far more than Z. So he gave E the shortest code (·) and Z a long one (––··). He figured this out by counting letters in a printer's type case. But the question is was his instinct right? Is this the best possible code?

The natural first question would be how do we even quantify how good a code (an assignment of letters to specific sequences of symbols) is. Rather than focusing on symbols themselves, we focus on what they transmit. To do this we first need to understand what letters even convey, the simplest answer would be meaning but more generally, if I say to you, 'There is a cat there', I'm conveying information. It is this information that shall be key to cracking this problem, and we try to maximise information per letter in order to be cost effective.

To quantify information, we consider what it actually tells you. Let's say I tell you the sun rose today, this gives you no information as it's inevitable. So events with probability 1 should carry 0 information. Additionally, a solar eclipse carries far more information than a sunrise because it is much rarer, so high probability events should give lower information. Furthermore, if I tell you it rained in London and a coin came up heads, for these independent events, as two separate facts, our total information should be the sum of each individual information, learning one tells you nothing about the other.

Now from the last property, that Information (I) of two events is their sum $I(p * q) = I(p) + I(q)$ (with p and q respective independent probabilities), should remind you of something. This function is exactly a logarithm and is in fact the only continuous function with this property since $I(1) = 0$ and $I(p^n) = nI(p)$. It is specifically by convention, $I(p) = -log_2(p)$ [1] for a probability p. The minus sign keeps it positive and the base 2 is because we measure information in bits. A fair coin coming up heads for example is worth exactly I(1/2) = 1 bit, but a rare event with probability 1/1000 carries $-\log_2(1000) \approx 10$ bits, ten times more information than a coin flip, matching our intuition. One final thing, if two or more variables aren't independent their combined entropy must be less than the sum of their individual entropies, e.g if I know it's stormy, I won't get much more information knowing it's raining. Their joint entropy is at most the

sum of individual entropies as knowing related things together tells you less than learning them independently would.

Now that we have the 'surprise' or information of each symbol, we can try quantifying the surprise of the alphabet as a whole. The expected information of the alphabet per letter is the probability of each symbol multiplied by the information of the symbol

$$E(X) = \sum_{i=1}^{n} P(x_i) I(x_i)$$

which is just

$$E(X) = -\sum_{i=1}^{n} P(x_i) \log_b P(x_i)$$

We will refer to this expected information as the Entropy. This Entropy is denoted by H (it wouldn't be my first choice of letter but information theory was late to the party, there weren't many left to choose from). Entropy therefore characterises the uncertainty of the source as a whole, for a fair coin $H = -(0.5 \log_2(0.5) + 0.5 \log_2(0.5)) = 1 \text{ bit}$

For a biased coin with p(H)=0.9, $H = -(0.9 \log_2(0.9) + 0.1 \log_2(0.1)) \approx 0.469 \text{ bits}$. Here the outcome is usually predictable, so the expected information of each flip is less. For the English alphabet as we use it (using real frequencies), H≈4.14 [2] per letter. This gives us a natural benchmark for any encoding scheme, bringing us back to our original question. Can any code actually achieve H bits per letter on average, and can we prove no code does better?

To understand the limits of coding, we consider long sequences of letters. Let's say we're tapping away at a typewriter randomly and we type n letters independently X_1,...,X_n their joint probability as they are independent is

$$p(X_1, \ldots, X_n) = \prod_{i=1}^{n} p(X_i) \text{ Product of individual probabilities}$$

Taking a log and dividing by n

$$-\frac{1}{n} \log_2 p(X_1, \ldots, X_n) = \frac{1}{n} \sum_{i=1}^{n} -\log_2 p(X_i)$$

The right side is the average information per letter of the sequence. We type these n letters with probabilities equal to their relative frequencies (e.g e more often than z) and so if we did this more and more we would expect to see maybe 50 Es out of 500, 1 Z etc. We can split this sum by each letter on the right side giving

$$= -\log_2 p(\mathrm{E}) \cdot \frac{count(\mathrm{E})}{n} + -\log_2 p(\mathrm{T}) \cdot \frac{count(\mathrm{T})}{n} + -\log_2 p(\mathrm{Z}) \cdot \frac{count(\mathrm{Z})}{n} \text{ etc}$$

If the sequence got longer and longer, the number of Es divided by total letters would just approach the probability of E (if E had probability 0.3 we would probably get around 300 Es out of 1000 ≈ 0.3) so the right side, as n gets larger, is

$$\approx -\log_2 p(\mathrm{E}) \cdot p(\mathrm{E}) + -\log_2 p(\mathrm{T}) \cdot p(\mathrm{T}) + -\log_2 p(\mathrm{Z}) \cdot p(\mathrm{Z}) \text{ etc}$$

This is just the entropy of the alphabet we defined above, meaning as n gets large $-\frac{1}{n}\log_2 p(X_1, \ldots, X_n) \to H(X)$, so the average information of a long string of letters approaches entropy.

We call a sequence of n letters "typical" if

$$\left| -\frac{1}{n}\log_2 p(x_1, \ldots, x_n) - H \right| < \varepsilon$$

for some small $\varepsilon > 0$. In other words, the average surprise (Information) of each symbol in the sequence gets very close to the entropy. This means its probability is close to 2^{-nH}, by rearranging this equation.

One way to understand this is with a biased coin with, say, p(Heads) = 0.9 (H=0.469). Here we expect 90 heads and 10 tails, and most sequences would be close to this ratio, yet each individual sequence is still very rare as specifically getting 86 heads 14 tails in some order is vanishingly unlikely. A typical sequence with around 90 heads would have probability roughly $2^{-nH} = 2^{-46.9}$, whereas a sequence of 90 tails would be atypical and much lower probability, as the average surprise (Information) of each symbol (due to the lower probability of tails) would be much higher than the entropy of the coin as a whole.

We call the collection of all typical sequences the typical set $\mathcal{T}_\varepsilon^{(n)}$. This may seem unintuitive, but if we want to optimise our code, we should focus on typical sequences e.g. words. This is because if you draw a sequence from a hat, it is overwhelmingly likely to be typical, e.g. a ratio of 9 heads per tail in the example above, anything atypical is orders of magnitude less likely as it requires unlikely outcomes to occur far more often than they should (A sequence of 50 Zs is technically possible, good luck finding it in a hat).

To show this we'll use the fact that the average information converges to the entropy

$$-\frac{1}{n}\log_2 p(X_1, \ldots, X_n) \xrightarrow{p} H$$

Convergence in probability (as it's uncertain, unlike 2x = 10) means that for any fixed $\varepsilon > 0$, the probability of the quantity differing from H by more than ε goes to zero as n grows:

$$P\left(\left| -\frac{1}{n}\log_2 p(X_1, \ldots, X_n) - H \right| \geq \varepsilon \right) \to 0$$

$= P(\text{NOT } \mathcal{T}_\varepsilon^{(n)}) < \varepsilon$ from our definition above

But the event inside this probability is exactly NOT being in $\mathcal{T}_\varepsilon^{(n)}$.

So if the probability of landing outside the typical set goes to zero, this means $P(\mathcal{T}_\varepsilon^{(n)}) \to 1$

Almost all sequences we will ever actually encounter are typical, meaning we don't need to care or worry about such weird sequences.

As mentioned before we want to see how compressed we can get our message, any lossless compression scheme is unique, different sequences must get different codewords. If two sequences map to the same codeword, you cannot tell which was sent. To distinguish between M sequences you need at least $\log_2 M$ bits. To see this, with k bits you can write at most 2^k distinct codewords, so to have each refer to a unique object $2^k \geq M \implies \log_2(2^k) \geq \log_2 M \implies k \geq \log_2 M$.

To know the minimum length/bits possible we need to know how many typical sequences there are. From above we know $P(\mathcal{T}_\varepsilon^{(n)}) > 1 - \varepsilon$ for large n.

Every sequence in the typical set has probability less than $2^{-n(H-\varepsilon)}$(by rearranging $\left|-\frac{1}{n}\log_2 p(x_1, \ldots, x_n) - H\right| < \varepsilon$), so:

$$1 - \varepsilon < P(\mathcal{T}_\varepsilon^{(n)}) \leq |\mathcal{T}_\varepsilon^{(n)}| \cdot 2^{-n(H-\varepsilon)}$$

Rearranging:

$$|\mathcal{T}_\varepsilon^{(n)}| \geq (1 - \varepsilon) \cdot 2^{n(H-\varepsilon)}$$

Since any code must assign distinct codewords to each typical sequence, and there are at least $(1 - \varepsilon) \cdot 2^{n(H-\varepsilon)}$ of them, any code needs at least this many bits.

$$\text{bits needed for sequence} \geq \log_2 |\mathcal{T}_\varepsilon^{(n)}| \geq \log_2 \left[(1 - \varepsilon) \cdot 2^{n(H-\varepsilon)}\right]$$

$$= n(H - \varepsilon) + \log_2(1 - \varepsilon)$$

Dividing by n, $L = \dfrac{\text{bits needed}}{n} \implies \text{L} = \text{average bits per letter}$

$$L \geq (H - \varepsilon) + \frac{\log_2(1 - \varepsilon)}{n}$$

And as n gets very very large, ε goes to 0 so

$L \geq H$

No code, not Morse, nor magic can compress fewer than H bits per letter, this is the lower limit. It's impossible to go lower. We can now compare Morse to the entropy of the English letters, around 4.14 [2] bits per letter. Morse's implementation using dots, dashes and crucially gaps (spaces), still achieves around 90% of the optimal efficiency for its own dot-dash framework [3], an exceptional achievement for counting letters from a type case.

Now that we know the minimum, the question is can we reach it, and surprisingly unlike counting to infinity we can get damn close. Let me introduce you to the Shannon Code.

It works like this, sort each symbol by probability largest first, then we assign each symbol a codeword of length $\ell_i = \lceil -\log_2 p_i \rceil$ ($\lceil x \rceil$ is the smallest integer equal to or bigger than x so $\lceil 2.94 \rceil = 3$). Now for each letter we calculate the sum of probabilities of letters more likely than it. The codeword for the symbol is then the first ℓ_i bits of the cumulative probability $F_i = \sum_{j<i} p_j$ written in binary.

Symbol	p_i	l_i	Sum from p_i to i-1	Sum to p_i binary	Code
S_1	0.36	2	0.0	0.00000⋯	00
S_2	0.18	3	0.36	0.01011⋯	010
S_3	0.18	3	0.54	0.10001⋯	100
S_4	0.12	3	0.72	0.10111⋯	101
S_5	0.09	4	0.84	0.11010⋯	1101
S_6	0.07	4	0.93	0.11101⋯	1110

[4]

Additionally a cool property is that no code word is a 'prefix' of another ('car' is a prefix of 'carpet') this means if you'd read '010' you know it's S_2 for example and that you don't need to continue on reading to see if it's part of another symbol. To prove this we consider two symbols i and j with $p_i > p_j$ and therefore cumulative probabilities satisfying $F_j \geq F_i + p_i$. Now since $\ell_i = \lceil -\log_2 p_i \rceil \geq -\log_2 p_i$, exponentiating gives $2^{-\ell_i} \leq p_i$.

Therefore, $F_j \geq F_i + p_i \geq F_i + 2^{-\ell_i}$. This says F_j and F_i differ by at least $2^{-\ell_i}$ which meaning their binary expansions must differ somewhere within the first ℓ_i binary places (S_1 differs from S_2 in the second place e.g. '0.00' vs '0.01') as each place differs by a power of two (not ten like we are used to). So the first ℓ_i bits of F_j cannot equal the first ℓ_i bits of F_i. Therefore codeword i is not a prefix of codeword j, as they differ in at least one of their starting digits, this means there are no prefixes in the entire code.

The beautiful consequence of having no prefixes, otherwise called a 'prefix code' (confusing name I know) is that you need no spaces since you know exactly when a code word begins or ends (you don't need to separate '110' and '1110' as neither is a prefix of another therefore does not need spaces).

Now that we know how it works, we need to show it's optimal.

Since $\lceil x \rceil \geq x$ and $\ell_i = \lceil -\log_2 p_i \rceil$

$\ell_i \geq -\log_2 p_i$, so the expected length L (probability of letter * length of codeword for each letter)

$$L = \sum_i p_i \ell_i \geq \sum_i p_i(-\log_2 p_i) = H$$

Additionally, as $\lceil x \rceil < x + 1$, we have

$\ell_i < -\log_2 p_i + 1$, so

$$L = \sum_i p_i \ell_i < \sum_i p_i(-\log_2 p_i + 1) = H + \sum_i p_i = H + 1$$

Therefore $H \leq L < H + 1$ meaning our average length is within 1 symbol of the entropy. If instead we coded blocks of letters of length n ($\text{ABB as an example of a 3 letter block and we would have } 26^3 \text{ blocks and therefore that many codewords}$) while there would be more codewords, the expected length per letter L_n could get as close to the entropy as we want. For large n as

$H(X_1, \ldots, X_n) \leq L_n < H(X_1, \ldots, X_n) + 1$, and as we assume independence, the Entropy of sequence is just n * Entropy per letter

$$nH(X) \leq L_n < nH(X) + 1$$

$$H(X) \leq \frac{L_n}{n} < H(X) + \frac{1}{n}$$

As n increases this gets closer and closer to entropy, as close as we want with the tradeoff of more codewords.

For English, with 4.14 [2] bits of entropy, we get at the very worst 5.14 bits per letter but in reality much closer to the entropy. So answering the question, Morse was not the best code but he was close, with his instincts being right on the money about codeword lengths, common things deserve short codes. These ideas, however, don't just refer to Morse code over telegram but work across all messages, languages and mediums, giving us a universal basis on which to build our messages. It is truly beautiful that with simple codes we can inch toward the universal boundary of information.

Sources:

[1] Khinchin, A.I. (1957). *Mathematical Foundations of Information Theory*. Dover Publications.

[2] Shannon, C.E. (1951). Prediction and Entropy of Printed English. *Bell System Technical Journal*. https://languagelog.ldc.upenn.edu/myl/Shannon1950.pdf

[3] Cook, J.D. (2017). *How efficiently does Morse code encode letters?* https://www.johndcook.com/blog/2017/02/08/how-efficient-is-morse-code/

[4] Lee, E., & Yoo, H. (2020). Example of Shannon coding. https://www.researchgate.net/figure/Example-of-Shannon-coding-The-code-length-li-of-each-symbol-can-be-obtained-from-the_tbl1_343563565