1. Explain what is meant by an optimal binary symbol code.

   An optimal binary symbol code is a symbol code over the alphabet $\{0,1\}$ where the expected length of a symbol (weighted by probability) is the entropy of the random variable.

2. (a) Suppose we wish to generate random numbers from a non-uniform distribution $\{p_0, p_1\} = \{0.99, 0.01\}$. Compare the following two techniques. Roughly how any random bits will each method use to generate a thousand samples from this sparse distribution?

   i. The standard method: use a standard random number generator to generate an integer between 1 and 2^{32}. Rescale the integer to $(0,1)$. Test whether this uniformly distributed random variable is less than 0.99 and emit a 0 or 1 accordingly.

   This method uses $32 \times 1000 = 32000$ random bits to generate 1000 samples from this sparse distribution.

   ii. Arithmetic Coding using the correct model fed with standard random bits.

   I provide two answers to this question: in the first, we *know* that 1000 samples will be generated, in the second we do not – but the *mean* number of samples we generate is 1000.

   Case 1: the number of bits required is 1000× the entropy of the random variable. $H(X) = 0.0807\ldots$, so $1000 \times H(X) = 80.79$ bits.

   Case 2: we introduce a third option – the `<EOS>` symbol # which has probability 0.001, so we are sending $Y = \{0.98901, 0.0999, 0.001\}$ and the number of bits sent is 1000× the entropy of Y. $H(Y) = 0.092\ldots$, so $1000 \times H(Y) = 92.1$ bits. So the expected number of bits is 92.1.

   (b) Use adaptive arithmetic coding (Laplace model) to encode in decimal the string `DEADBEEF#` (where # is the end-of-string symbol). The 5-symbol source alphabet is `A`, `B`, `D`, `E`, `F`. Use a fixed probability of 0.05 for the end-of-string symbol.

   Using Laplace's rule, with F_i as the number of occurrences of symbol i; and p_i as the probability of seeing i, we have:

$$p_i = (1 - p_\#) \cdot \frac{F_i + 1}{\sum_i (F_i + 1)}$$

   I assume that the probability mass assigned to # is $0.95 - -1.0$.

   I wrote a program which does this using 1000 decimal places. It computed that the probability was $[0.5020770999\ldots, 0.5020771198\ldots)$. The shortest bit sequence in this range is `1000000010001000001`; which represents $0.50207710\ldots$.

```
from collections import Counter, namedtuple
import decimal
from decimal import Decimal

decimal.getcontext().prec = 1000

prob = namedtuple('Probability', 'p')
cumprob = namedtuple('ProbabilityRange', ['lo', 'hi'])


def laplace_probs(F, p_eos, characters):
    total = Decimal(sum(F.values()))
    seen = Decimal(0)
    probs = {}
```

```
    cumprobs = {}
    for char in characters:
        probs[char] = prob(F[char] / total * (1 - p_eos))
        p_lo = seen / total * Decimal(0.95)
        seen += F[char]
        p_hi = seen / total * Decimal(0.95)
        cumprobs[char] = cumprob(p_lo, p_hi)
    probs['#'] = p_eos
    cumprobs['#'] = cumprob(Decimal(0.95), Decimal(1.))
    return probs, cumprobs


def code(sequence: str,
         characters=('A', 'B', 'D', 'E', 'F'),
         p_eos: float = 0.05):
    p_eos = Decimal(p_eos)
    lo, hi = Decimal(0), Decimal(1)
    F = Counter(characters)

    for char in sequence:
        probs, cumprobs = laplace_probs(F, p_eos, characters)

        lo, hi = lo + (hi - lo) * cumprobs[char].lo, lo + (
            hi - lo) * cumprobs[char].hi

        F[char] += 1

    low, high = lo, hi

    i = Decimal(0.5)
    rep = ''
    while lo != 0:
        if lo <= i < hi:
            rep += '1'
            return low, high, rep
        elif i < lo and i < hi:
            lo -= i
            hi -= i
            rep += '1'
        else:
            rep += '0'
        i /= 2

    return low, high, rep


if __name__ == '__main__':
    lo, hi, rep = code('DEADBEEF#')
    print(f'{lo}\n{hi}\n{rep}')
```

✓ Fantastic!

3. (a) Encode the string `000000000000100000000000` using the basic Lempel-Ziv algorithm presented in lectures. Use a 3-bit fixed-length dictionary; ignore termination of the string.

   Start by initialising the dictionary to $\{0 \mapsto 000, 1 \mapsto 001\}$.

Remaining Message	Longest Match	Code to add to Dictionary
000000000000100000000000	0 ↦ 000	00 ↦ 010
000000000001000000000000	00 ↦ 010	000 ↦ 011
000000001000000000000	000 ↦ 011	0000 ↦ 100
000001000000000000	0000 ↦ 100	00000 ↦ 101
00100000000000	00 ↦ 010	001 ↦ 110
100000000000	1 ↦ 001	10 ↦ 111
00000000000	00000 ↦ 101	-
000000	00000 ↦ 101	-
0	0 ↦ 000	-

Table 1: Table showing how to encode a sequence using Lempel-Ziv

Good. Explicitly give output.

(b) Using a variable-length dictionary (max. 3-bits) initialised to $0 = 0, 1 = 1$ and ignoring end-of-string markers, decode the LZ-encoded string: 00110010001110100010.

I interpret a "variable-length dictionary" to be a dictionary which is expanded dynamically when it is filled up; to a maximum length of 3 bits.

Yep, sorry for the vagueness.

I will use the "sliding window method" to decode the code.

Matched	Updates to Dictionary	Add to Dictionary
0 0	-	10 ↦ 0_

Table 2: Table showing the decoding with a dictionary of length 1

The dictionary is now full; so it is resized. This means that it is extended by increasing the size of every key in the dictionary by 1. So, the dictionary is now $\{00 \mapsto 0, 01 \mapsto 1, 10 \mapsto 0\_\}$

Matched	Updates to Dictionary	Add to Dictionary
01 1	10 ↦ 01	11 ↦ 1_
10 01	11 ↦ 10	100 ↦ 01_

Table 3: Table showing the decoding with a dictionary of length 2

The dictionary is now full; so it is resized. This means that it is extended by increasing the size of every key in the dictionary by 1. So, the dictionary is now $\{000 \mapsto 0, 001 \mapsto 1, 010 \mapsto 01, 11 \mapsto 10, 100 \mapsto 01\_\}$

Matched	Updates to Dictionary	Add to Dictionary
010 01	100 ↦ 010	101 ↦ 01_
001 1	101 ↦ 011	110 ↦ 1_
110 11	110 ↦ 11	111 ↦ 11_
100 010	111 ↦ 110	-
010 01	-	-

Table 4: Table showing the decoding with a dictionary of length 3

We can now read off the message that has been decoded: 01010111101001.

Awesome!

(c) Give examples of simple sources that have low entropy but would not be compressed well by the Lempel-Ziv algorithm.

I propose sending the positions of a point cloud for a scan of a perfectly flat planar object such as a wall. The data is of the form (x, y, z). Since the surface is flat, there is very little uncertainty on the value of z after x has been seen. However, Lempel-Ziv will fail to exploit this since there is very little *exact* repetition at a bit-level.

Good.

I propose sending the value of π. After the first few bits have been sent, it's very obvious that this *is* π *i.e.* after receiving $3.14159\ldots$ any human (knowing the value of π) would recognise what was being sent and receive a very low surprisal from the next digits. However, since π is a transcendental number, the individual bits have no meaningful repetition. This means that an encoding by the Lempel-Ziv algorithm (or any simple encoder) would not exploit *any* redundancy despite the fact that the surprisal of each bit would tend to zero. The same premise applies for sending any "known" message.

Nice! You could formalise this through Kolmogorov complexity.

4. Consider a $(7, 4)$ Hamming Code which maps $k = 4$ information bits to a length $n = 7$ codeword. Assume 0 and 1 are equiprobable in the input data.

   Use the convention used by the rest of the world, not the one presented in lectures. The transmitted codeword is

   $$[b_1, b_2, b_3, b_4, b_5, b_6, b_7]$$

   where b_3, b_5, b_6 and b_7 are source bits and

   $$\begin{aligned} b_4 &= b_5 \oplus b_6 \oplus b_7 \\ b_2 &= b_3 \oplus b_6 \oplus b_7 \\ b_1 &= b_3 \oplus b_5 \oplus b_7 \end{aligned}$$

   (a) Suppose a codeword is transmitted over a binary symmetric channel (BSC) and the received codeword is $r = [1, 1, 0, 1, 0, 1, 1]$. Decode the received sequence to a codeword.

   By recomputing the parity bits from the received message, we get $b_4 = 0 \oplus 1 \oplus 1 = 0$, $b_2 = 0 \oplus 1 \oplus 1 = 0$, $b_1 = 0 \oplus 0 \oplus 1 = 1$. We see that b_1, b_2 and b_4 are all different to the received message. Therefore bit $4 + 2 + 1 = 7$ has been corrupted. So the error correction will flip it and return 0010.

   0001.

   (b) Calculate the probability of block error p_B of the $(7, 4)$ Hamming code as a function of the bit error p and show that to leading order it goes as $21p^2$.

   Let $P(i)$ be the probability of exactly i errors:

   $$\begin{aligned} p_B &= 1 - P(0) - P(1) \\ &= 1 - (1-p)^7 - 7 \cdot p(1-p)^6 \\ &= 1 - 1 + 7p - 21p^2 - 7p + 42p^2 + \mathcal{O}(p^3) \\ &= 21p^2 + \mathcal{O}(p^3) \end{aligned}$$

   (c) If the $(7, 4)$ Hamming code can correct any one bit error, might there be a $(14, 8)$ code that can correct any two errors?

   No, there is not. I provide an information-theoretic proof.

   To correct 2 bits in an n bit code, we require enough parity bits to tell us which bits have been corrupted. So, we require $\lg(n+1) + \lg n$ bits of information in addition to the bits in the message. The first $\lg(n+1)$ bits are required to indicate what the first bit that was corrupted is. The $\lg n$ bits are required to indicate which the second bit that was corrupted is.

For a code of length 14, this means we require $\lg 15 + \lg 14 = 7.7$ bits of redundancy. However the proposed scheme only has 6 bits of redundancy! Therefore no such code can exist.

Good. Nicely done :-)

5. Consider using the repetition code R_5 to encode binary input symbols for transmission through a binary symmetric channel with $f = 0.3$. Assuming $p_0 < 0.5$, find the maximum value of p_0 for which the optimal decoder's rule is not simply "pick the majority vote".

   The first message which will be translated not by majority vote is a message containing three 0 and two 1.

$$\frac{P(1 \mid \text{message})}{P(0 \mid \text{message})} = \frac{P(\text{message} \mid 1)}{P(\text{message} \mid 0)} \cdot \frac{P(1)}{P(0)} \tag{1}$$

   In order to select 1, we have that the decision ratio must be greater than 1

$$1 < \frac{P(\text{message} \mid 1)}{P(\text{message} \mid 0)} \cdot \frac{P(1)}{P(0)} \tag{2}$$

$$1 < \frac{10 \cdot f^3 \cdot (1-f)^2}{10 \cdot f^2 \cdot (1-f)^3} \cdot \frac{1-p_0}{p_0} \tag{3}$$

$$1 < \frac{f}{1-f} \cdot \frac{1-p_0}{p_0} \tag{4}$$

$$p_0 - p_0 f < f - p_0 f \tag{5}$$

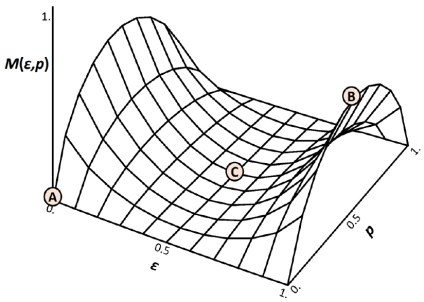
$$p_0 < f \tag{6}$$

   ✓

   So, if we have that $p_0 < f = 0.3$, then we have that the optimal rule is *not* majority vote.

   (universally) ✓ Good.

6. A binary symmetric channel receives as input a bit whose values $\{0, 1\}$ have probabilities $\{p, 1-p\}$, but in either case, a transmission error can occur with probability ϵ which flits the bit. The surface plot below describes the mutual information of this channel as a function $M(\epsilon, p)$ of these probabilities.



   (a) At the point marked A, the error probability is $\epsilon = 0$. Why then is the channel mutual information minimal in this case: $M(\epsilon, p) = 0$?

   $p = 0$. Thus receiving a 0 is certain and so conveys no information. We have $I(X;Y) = H(X) - H(X \mid Y) = 0 - 0 = 0$. ✓

   (b) At the point marked B, an error always occurs ($\epsilon = 1$). Why then is the channel mutual information maximal in this case: $M(\epsilon, p) = 1$. At this point, the error probability is 1.

   Error probability of 1 means that on receiving symbol σ, we are guaranteed the original message was $1 - \sigma$. So on receiving a message we have no uncertainty about its value: i.e. $H(X \mid Y) = 0$. So we have $I(X;Y) = H(X) - H(X \mid Y) = H(X)$. Since $p = 0.5$, we have $H(X) = 1$ and so $I(X;Y) = 1$. ✓

(c) At the point marked C, the input bit values are equiprobable ($p = 0.5$), so the symbol source has maximal entropy. Why then is the channel mutual information in this case $M(\epsilon, p) = 0$.

$H(Y \mid X) = 1$: knowing the output tells us nothing about the input! So, even though we have $H(X) = H(Y) = 1$, we have $I(X;Y) = H(Y) - H(Y \mid X) = 1 - 1 = 0$.

Fantastic work!