src\arithmetic_coder.py

```python
from fractions import Fraction
from collections import defaultdict
from bitarray import bitarray
import struct

def build_freqs(symbols):
    # symbols: iterable of ints (0..7)
    freq = [0]*8
    for s in symbols:
        freq[s] += 1
    # ensure nonzero for stable CDF (add 1 to any zero freq)
    for i in range(8):
        if freq[i] == 0:
            freq[i] = 1
    total = sum(freq)
    # cumulative
    cdf = [0]*(len(freq)+1)
    c = 0
    for i,f in enumerate(freq):
        cdf[i] = c
        c += f
    cdf[len(freq)] = c
    return freq, cdf, total

def arithmetic_encode(symbols):
    # Build freq model from symbols
    freq, cdf, total = build_freqs(symbols)
    # arithmetic encode using Fractions
    low = Fraction(0,1)
    high = Fraction(1,1)
    for s in symbols:
        range_ = high - low
        high = low + range_ * Fraction(cdf[s+1], total)
        low  = low + range_ * Fraction(cdf[s], total)
    # choose a binary fraction inside [low, high)
    # produce bits by repeatedly doubling low until it >= high
    bits = bitarray()
    x = low
    # produce enough bits: until x < high and either enough precision produced
    # Stop when low and high have different binary prefixes
    max_bits = 1024  # safety cap
    while True:
        if x >= 1:
            bits.append(1)
            x -= 1
        else:
            bits.append(0)
        # advance low/high by doubling
        low = low * 2
        high = high * 2
        if low >= 1:
```

```
            low -= 1
        if high >= 1:
            high -= 1
        if len(bits) > 0:
            # rebuild fractions represented by current bit sequence as binary fraction
            # value_v is the rational represented by bits
            value_v = Fraction(0,1)
            pow2 = Fraction(1,1)
            for b in bits:
                pow2 *= 2
                if b:
                    value_v += Fraction(1, pow2)
            # if value_v >= low and value_v < high then we can stop
            if (value_v >= low) and (value_v < high):
                break
        if len(bits) >= max_bits:
            break
    # pack header (frequencies) as 8 uint32 little-endian
    header = b''.join(struct.pack('<I', f) for f in freq)
    # pack bit length as uint32 big-endian
    bitlen = len(bits)
    header2 = struct.pack('>I', bitlen)
    # pack bits to bytes
    bitbytes = bits.tobytes()
    return header + header2 + bitbytes

def arithmetic_decode(blob):
    import struct
    # read 8*4 bytes freq header
    if len(blob) < 8*4 + 4:
        raise ValueError("Invalid encoded blob")
    header = blob[:8*4]
    freq = []
    for i in range(8):
        (f,) = struct.unpack('<I', header[i*4:(i+1)*4])
        freq.append(f)
    rest = blob[8*4:]
    (bitlen,) = struct.unpack('>I', rest[:4])
    bitbytes = rest[4:]
    bits = bitarray()
    bits.frombytes(bitbytes)
    # trim to bitlen
    bits = bits[:bitlen]
    # build cdf
    total = sum(freq)
    cdf = [0]*(len(freq)+1)
    c = 0
    for i,f in enumerate(freq):
        cdf[i] = c
        c += f
    cdf[len(freq)] = c
    # reconstruct encoded_value fraction from bits
    encoded_value = Fraction(0,1)
    pow2 = Fraction(1,1)
```

```
    for b in bits:
        pow2 *= 2
        if b:
            encoded_value += Fraction(1, pow2)
    # decoding loop (Algorithm 2)
    low = Fraction(0,1)
    high = Fraction(1,1)
    symbols = []
    max_symbols = 1000000
    for _ in range(max_symbols):
        range_ = high - low
        # compute scaled value = (encoded_value - low) / range_
        scaled = (encoded_value - low) / range_
        # find symbol j s.t. CDF(j) <= scaled*total < CDF(j+1)
        # scaled is Fraction in [0,1)
        target = scaled * total
        # find j
        j = None
        for idx in range(len(freq)):
            if target >= cdf[idx] and target < cdf[idx+1]:
                j = idx
                break
        if j is None:
            break
        symbols.append(j)
        # update interval
        high = low + range_ * Fraction(cdf[j+1], total)
        low  = low + range_ * Fraction(cdf[j], total)
        # termination heuristic: if low==high break
        if high - low <= 0:
            break
        # If the decoded symbols so far, when encoded, would give an interval that
contains encoded_value,
        # we continue until bit precision exhausted. We cap by bitlen.
        # Stop condition: if we've decoded more symbols than some bound and the next
refinement gives no change, break.
        if len(symbols) > 0 and len(symbols) > (bitlen*2):
            break
    return symbols
```