02-Binary_Compression_Prototype.py

python

```python
# Binary Compression Prototype v1.0
# Run: python this_file.py --input audio.bin --output compressed.clr
# Requires: No extras—pure Python for demo.

import argparse
import struct  # For bit packing
from collections import defaultdict, deque
import random  # For Evotar seed

class BackRef:
    def __init__(self, dist, length):
        self.dist = dist
        self.length = length

class Literal:
    def __init__(self, byte):
        self.byte = byte

def find_longest_match(stream, pos, window_size=32768):
    """Simple LZ77 match finder."""
    max_len = 0
    max_dist = 0
    start = max(0, pos - window_size)
    for length in range(3, min(258, len(stream) - pos) + 1):  # Max 258 per RFC1951
        for dist in range(1, min(pos - start, 32768) + 1):
            if stream[pos:pos+length] == stream[pos-dist:pos-dist+length]:
                if length > max_len:
                    max_len = length
                    max_dist = dist
    return max_len, max_dist

def build_huffman_tree(symbols):
    """Canonical Huffman tree from freqs."""
    freq = defaultdict(int)
    for sym in symbols:
        freq[sym] += 1
    # Sort + build (simplified—use heapq for prod)
    sorted_syms = sorted(freq.items(), key=lambda x: x[1])
    # Placeholder: Equal weights for demo
    codes = {sym: i for i, sym in enumerate(sorted_syms[:256])}  # 8-bit syms
    return codes

def huffman_encode(blocks, codes):
```

```
    """Encode blocks to bitstring."""
    bitstream = []
    for block in blocks:
        if isinstance(block, Literal):
            code = codes.get(block.byte, 0)
        else:
            code = codes.get((block.dist, block.length), 0)
        bitstream.extend(format(code, '08b'))  # 8-bit for demo
    return ''.join(bitstream)

def evotar_prune(codes, generations=5, population=50):
    """Simple genetic prune: Mutate codes, select best entropy."""
    def entropy(c):
        return sum(c.values()) / len(c)  # Mock low-entropy selector
    population = [codes.copy() for _ in range(population)]
    for _ in range(generations):
        for indiv in population:
            if random.random() < 0.1:  # Mutate 10%
                key = random.choice(list(indiv.keys()))
                indiv[key] += random.randint(-1, 1)
        population = sorted(population, key=entropy)[:population//2]  # Elitism
    return population[0]  # Best

def encode_clarity_v28(input_stream):
    """Full encode (from pseudocode)."""
    lz77_blocks = []
    i = 0
    while i < len(input_stream):
        match_len, match_dist = find_longest_match(input_stream, i)
        if match_len >= 3:
            lz77_blocks.append(BackRef(match_dist, match_len))
            i += match_len
        else:
            lz77_blocks.append(Literal(input_stream[i]))
            i += 1

    codes = build_huffman_tree([b.byte if isinstance(b, Literal) else (b.dist, b.length) for b
    evolved_codes = evotar_prune(codes)
    huff_bits = huffman_encode(lz77_blocks, evolved_codes)

    # Pack to bytes
    packed = bytearray()
    for i in range(0, len(huff_bits), 8):
        byte_str = huff_bits[i:i+8]
        packed.append(int(byte_str or '0'*8, 2))
    return bytes(packed)

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
```

```
parser.add_argument("--input", required=True)
parser.add_argument("--output", required=True)
args = parser.parse_args()

with open(args.input, "rb") as f:
    data = f.read()

compressed = encode_clarity_v28(data)
with open(args.output, "wb") as f:
    f.write(compressed)

print(f"Compressed {len(data)} -> {len(compressed)} bytes ({len(compressed)/len(data)*100:
```