#!/usr/bin/env python

import struct

import wave

import numpy as np

from scipy.fftpack import idct

def read\_compressed(filename):

"""Read the compressed file and parse the header."""

with open(filename, 'rb') as f:

data = f.read()

# Parse header:

# block\_size (int32), pad (int32), quant\_step (float64), nblocks (int32), padding\_bits(int32), huff\_entries(int32)

offset = 0

block\_size = struct.unpack\_from('<i', data, offset)[0]

offset += 4

pad = struct.unpack\_from('<i', data, offset)[0]

offset += 4

quant\_step = struct.unpack\_from('<d', data, offset)[0]

offset += 8

nblocks = struct.unpack\_from('<i', data, offset)[0]

offset += 4

padding\_bits = struct.unpack\_from('<i', data, offset)[0]

offset += 4

huff\_entries = struct.unpack\_from('<i', data, offset)[0]

offset += 4

# Read Huffman table entries

# Each entry: symbol(int16), code\_len(int16), code\_str(ascii)

huffman\_table = {}

for \_ in range(huff\_entries):

symbol = struct.unpack\_from('<h', data, offset)[0]

offset += 2

code\_len = struct.unpack\_from('<h', data, offset)[0]

offset += 2

code\_str = data[offset:offset+code\_len].decode('ascii')

offset += code\_len

huffman\_table[symbol] = code\_str

# The rest is compressed bitstream

compressed\_bitstream = data[offset:]

return block\_size, pad, quant\_step, nblocks, padding\_bits, huffman\_table, compressed\_bitstream

def build\_code\_to\_symbol\_map(huffman\_table):

"""Invert the huffman\_table from symbol->code to code->symbol."""

# huffman\_table: {symbol: code\_str}

code\_to\_symbol = {code: sym for sym, code in huffman\_table.items()}

return code\_to\_symbol

def bytes\_to\_bitstring(byte\_data):

"""Convert bytes to a bitstring."""

return ''.join(bin(byte\_val)[2:].zfill(8) for byte\_val in byte\_data)

def huffman\_decode(bitstring, code\_to\_symbol, padding\_bits):

"""Decode the bitstring using the Huffman code\_to\_symbol map."""

# Remove the extra padding bits at the end

if padding\_bits > 0:

bitstring = bitstring[:-padding\_bits]

# Since Huffman codes are prefix-free, we can decode symbol by accumulating bits

decoded\_symbols = []

current\_code = ''

# To speed up lookups, note that since codes differ in length, we must check incrementally.

# We'll accumulate bits in current\_code and check if it forms a complete symbol.

# Because codes are prefix-free, any match we find is a complete symbol.

code\_set = set(code\_to\_symbol.keys())

# Maximum code length could help performance, but not strictly needed.

# We'll just do a loop lookup.

for bit in bitstring:

current\_code += bit

if current\_code in code\_to\_symbol:

decoded\_symbols.append(code\_to\_symbol[current\_code])

current\_code = ''

return decoded\_symbols

def uniform\_dequantize(qdata, step):

return qdata.astype(np.float64) \* step

if \_\_name\_\_ == "\_\_main\_\_":

# Read compressed file and parse header

block\_size, pad, quant\_step, nblocks, padding\_bits, huffman\_table, compressed\_bitstream = read\_compressed('compressed')

# Invert Huffman table to code->symbol

code\_to\_symbol = build\_code\_to\_symbol\_map(huffman\_table)

# Convert compressed data to a bitstring

bitstring = bytes\_to\_bitstring(compressed\_bitstream)

# Decode Huffman

quantized\_list = huffman\_decode(bitstring, code\_to\_symbol, padding\_bits)

quantized\_array = np.array(quantized\_list, dtype=np.int16)

# Reshape to (nblocks, block\_size)

quantized\_blocks = quantized\_array.reshape(nblocks, block\_size)

# Dequantize

dct\_blocks = uniform\_dequantize(quantized\_blocks, quant\_step)

# Inverse DCT (type=2, norm='ortho')

# Matches the DCT settings from compress.py

time\_blocks = idct(dct\_blocks, type=2, norm='ortho', axis=1)

# Reassemble the full signal

reconstructed = time\_blocks.flatten()

# Remove padding samples

if pad > 0:

reconstructed = reconstructed[:-pad]

# Convert back to 16-bit PCM

# Clip to avoid overflow

reconstructed = np.clip(reconstructed, -1.0, 1.0)

int\_samples = (reconstructed \* (2\*\*15)).astype('<h')

# Write to out.wav

# Use the original parameters from step.wav: 1 channel, 16-bit, 44100 Hz

# The instructions specify these parameters remain consistent.

with wave.open('out.wav', 'wb') as f:

f.setnchannels(1)

f.setsampwidth(2)

f.setframerate(44100)

f.writeframes(int\_samples.tobytes())

print("Decompression complete. 'out.wav' created.")