#!/usr/bin/env python3

import wave

import struct

import numpy as np

from scipy.fftpack import dct

import sys

import heapq

# =============================================================================

# Helper Functions

# =============================================================================

def read\_wav(filename):

"""Read a mono 16-bit PCM wav file and return samples as a NumPy float array."""

with wave.open(filename, 'r') as f:

params = f.getparams()

nframes = params.nframes

raw\_bytes = f.readframes(nframes)

samples = np.frombuffer(raw\_bytes, dtype='<h').astype(np.float64)

# Normalize samples to [-1,1]

samples /= 2\*\*15

return samples, params

def write\_compressed(filename, header\_data, bitstream):

"""Write the compressed header and bitstream to a binary file."""

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

# Write header data first

f.write(header\_data)

# Write the compressed bitstream

f.write(bitstream)

def block\_partition(samples, block\_size):

"""Partition the signal into blocks of length block\_size.

If not divisible, pad with zeros at the end."""

n = len(samples)

pad = 0

if n % block\_size != 0:

pad = block\_size - (n % block\_size)

samples = np.concatenate((samples, np.zeros(pad)))

# Reshape into blocks

blocks = samples.reshape(-1, block\_size)

return blocks, pad

def block\_dct(blocks):

"""Apply DCT-II to each block along its rows."""

# type=2, norm='ortho' gives a proper orthonormal DCT

return dct(blocks, type=2, norm='ortho', axis=1)

def block\_idct(dct\_blocks):

"""Inverse DCT for testing or reference (not needed for compression step)."""

from scipy.fftpack import idct

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

def uniform\_quantize(data, step):

"""Uniform scalar quantization: round(data/step)."""

return np.round(data / step).astype(np.int16)

def uniform\_dequantize(qdata, step):

"""Inverse of uniform quantization: qdata \* step."""

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

# =============================================================================

# Huffman Coding

# =============================================================================

class HuffmanNode:

def \_\_init\_\_(self, symbol=None, freq=0):

self.symbol = symbol

self.freq = freq

self.left = None

self.right = None

def \_\_lt\_\_(self, other):

return self.freq < other.freq

def build\_huffman\_tree(symbols\_freq):

"""Build a Huffman tree given a dictionary of symbol frequencies."""

heap = []

for sym, freq in symbols\_freq.items():

node = HuffmanNode(sym, freq)

heapq.heappush(heap, node)

if len(heap) == 1:

# Edge case: only one symbol

node = heapq.heappop(heap)

root = HuffmanNode()

root.left = node

return root

while len(heap) > 1:

node1 = heapq.heappop(heap)

node2 = heapq.heappop(heap)

parent = HuffmanNode()

parent.freq = node1.freq + node2.freq

parent.left = node1

parent.right = node2

heapq.heappush(heap, parent)

return heapq.heappop(heap)

def build\_huffman\_codes(root):

"""Build Huffman codes from Huffman tree."""

codes = {}

def traverse(node, code):

if node.symbol is not None:

codes[node.symbol] = code

return

if node.left:

traverse(node.left, code + '0')

if node.right:

traverse(node.right, code + '1')

traverse(root, "")

return codes

def huffman\_encode(data, codes):

"""Encode the data using given Huffman codes. Return a bitstring."""

return ''.join(codes[sym] for sym in data)

def pad\_bitstring(bitstring):

"""Pad bitstring to a multiple of 8 bits."""

extra\_bits = (8 - (len(bitstring) % 8)) % 8

bitstring += '0' \* extra\_bits

return bitstring, extra\_bits

def bitstring\_to\_bytes(bitstring):

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

return bytes(int(bitstring[i:i+8], 2) for i in range(0, len(bitstring), 8))

# =============================================================================

# Main Compression Steps

# =============================================================================

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

# Input parameters (tune these)

block\_size = 1024

quant\_step = 0.0005 # Adjust for desired quality. Smaller step = less error, larger file.

# Read the input wav

samples, params = read\_wav('step.wav')

# Partition into blocks

blocks, pad = block\_partition(samples, block\_size)

# Apply DCT

dct\_blocks = block\_dct(blocks)

# Quantize all DCT coefficients

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

quantized\_flat = quantized.flatten()

# Build frequency table for Huffman coding

symbol\_freq = {}

for val in quantized\_flat:

symbol\_freq[val] = symbol\_freq.get(val, 0) + 1

# Build Huffman tree and codes

root = build\_huffman\_tree(symbol\_freq)

codes = build\_huffman\_codes(root)

# Encode quantized data using Huffman codes

bitstring = huffman\_encode(quantized\_flat, codes)

# Pad and convert to bytes

bitstring, padding\_bits = pad\_bitstring(bitstring)

compressed\_data = bitstring\_to\_bytes(bitstring)

# Prepare header:

# We'll store:

# [block\_size (int32), pad (int32), quant\_step (float64), nblocks (int32), Huffman table size, Huffman table entries...]

# Then the compressed bitstream.

nblocks = quantized.shape[0]

# Store Huffman code table:

# We'll store it as: number\_of\_entries, then for each entry: symbol (int16), code\_length (int16), code\_bits (as string)

# Convert code strings to a binary representation for compactness.

huffman\_table = []

for sym, c in codes.items():

# sym is int16

code\_len = len(c)

# We'll just store the code as ASCII '0'/'1' characters.

# (You can improve this by packing bits)

huffman\_table.append((sym, code\_len, c))

header = bytearray()

# block\_size -> int32

header += struct.pack('<i', block\_size)

# pad -> int32

header += struct.pack('<i', pad)

# quant\_step -> float64

header += struct.pack('<d', quant\_step)

# nblocks -> int32

header += struct.pack('<i', nblocks)

# padding\_bits -> int32

header += struct.pack('<i', padding\_bits)

# number\_of\_entries in Huffman table

header += struct.pack('<i', len(huffman\_table))

# Each entry: symbol (int16), code\_len(int16), code as bytes (ASCII)

for (sym, code\_len, code\_str) in huffman\_table:

header += struct.pack('<h', sym)

header += struct.pack('<h', code\_len)

# Store code\_str as raw ASCII

header += code\_str.encode('ascii')

# Write everything out

write\_compressed('compressed', header, compressed\_data)

print("Compression complete. 'compressed' file created.")

print(f"Original size: {samples.nbytes} bytes")

print(f"Compressed size: {len(header) + len(compressed\_data)} bytes")