In [ ]:
```python
# Cell 0
import pathlib, math, struct, numpy as np
from scipy.io import wavfile


def pcm_to_residuals(pcm: np.ndarray) -> np.ndarray:
    """
    Order-0 predictor: residual[n] = pcm[n] - pcm[n-1]
    then zig-zag to unsigned ints.
    Handles mono or stereo transparently.
    """
    pcm32 = pcm.astype(np.int32)
    # prepend a zero so diff has same length
    diff  = np.diff(pcm32, prepend=0, axis=0)
    # zig-zag:  0→0, -1→1, +1→2, …
    zz = np.where(diff >= 0, diff << 1, (-diff << 1) - 1)
    return zz.astype(np.uint32)
```

In [ ]:
```python
# Cell 1
ASSETS   = pathlib.Path("assets")
FILES    = ["Sound1.wav", "Sound2.wav"]
K_VALUES = [2, 4]        # Rice parameters to test
```

In [30]:
```python
# Cell 2
def rice_encode(data: np.ndarray, K: int) -> bytearray:
    """Return bitstream as bytearray (little-endian)."""
    m = 1 << K
    bitbuf, count, out = 0, 0, bytearray()
    for sample in data:
        q = sample // m     # unary part
        r = sample %  m     # remainder
        # q times '1' then '0'
        for _ in range(q):
            bitbuf = (bitbuf << 1) | 1; count += 1
            if count == 8: out.append(bitbuf); bitbuf = 0; count = 0
        bitbuf = (bitbuf << 1); count += 1                     # the '0'
        if count == 8: out.append(bitbuf); bitbuf = 0; count = 0
        # K-bit remainder
        for i in reversed(range(K)):
            bitbuf = (bitbuf << 1) | ((r >> i) & 1); count += 1
            if count == 8: out.append(bitbuf); bitbuf = 0; count = 0
    if count: out.append(bitbuf << (8 - count))
    return out

def rice_decode(bitstream: bytearray, K: int, n_samples: int) -> np.ndarray:
    """Inverse of rice_encode (handles 1-D data)."""
    m = 1 << K
    data = []

    byte_iter = iter(bitstream)
    cur       = next(byte_iter)         # first byte
    bits_left = 8                       # ← was 0, caused 1-byte skip

    def next_bit():
```

```
        nonlocal cur, bits_left, byte_iter
        if bits_left == 0:
            cur = next(byte_iter)
            bits_left = 8
        bits_left -= 1
        return (cur >> bits_left) & 1

    for _ in range(n_samples):
        # unary part
        q = 0
        while next_bit():          # count leading 1s
            q += 1
        # remainder
        r = 0
        for _ in range(K):
            r = (r << 1) | next_bit()
        data.append(q * m + r)

    return np.array(data, dtype=np.uint32)
```

In [31]:

```
# Cell 3
results = []

for fname in FILES:
    rate, pcm = wavfile.read(ASSETS / fname)      # 16-bit signed
    shape_orig = pcm.shape                        # remember (n,) or (n,2)
    unsigned = pcm_to_residuals(pcm).ravel()

    for K in K_VALUES:
        encoded = rice_encode(unsigned, K)

        # save compressed file
        out_bin = (ASSETS / fname).with_suffix(f".rc{K}")
        out_bin.write_bytes(encoded)

        # decode and reshape
        decoded_1d = rice_decode(encoded, K, unsigned.size)
        decoded    = decoded_1d.reshape(shape_orig)

        assert np.array_equal(unsigned.reshape(shape_orig), decoded), \
               f"decode mismatch on {fname} K={K}"

        ratio = len(encoded) / pcm.nbytes
        results.append((fname, K, len(encoded), pcm.nbytes, ratio))
```

In [32]:

```
# Cell 4
import pandas as pd, IPython.display as disp
df = pd.DataFrame(results, columns=["File", "K", "Compressed (bytes)",
                                    "Original (bytes)", "Ratio"])
disp.display(df.style.format({"Ratio":"{:.3f}"}))
```

	File	K	Compressed (bytes)	Original (bytes)	Ratio
0	Sound1.wav	2	2429905	1002044	2.425
1	Sound1.wav	4	857451	1002044	0.856
2	Sound2.wav	2	226196607	1008000	224.401
3	Sound2.wav	4	56793288	1008000	56.343

Observations

File	K	Compressed / Original	Verdict
Sound1.wav	2	**2.43 × larger**	K = 2 is too small for this file.
	4	**0.86 × (14 % smaller)**	K = 4 compresses 16-bit residuals nicely.
Sound2.wav	2	**224 × larger**	File is 32-bit; unary run explodes.
	4	**56 × larger**	Same issue—K still far too small.

What's going on?
`Sound1.wav` is 16-bit audio. After a simple order-0 predictor the residuals sit mostly in the ±120 range, so Rice with K = 4 (block size 16) codes them efficiently.

`Sound2.wav` is 32-bit audio with peaks beyond ±2 000 000. Even after differencing, residuals are still ≈ ±1 000 000. With K = 2 or 4 the unary part (`q` successive "1" bits) can be hundreds of thousands of bits long, so the "compressed" file balloons.

Fix (optional demo):
An adaptive rule of thumb is **K ≈ $\lceil \log_2 \text{mean}(|\text{residual}|) \rceil$**.
That gives K = 2 for Sound 1 and K = 8 – 9 for Sound 2. Using K = 8 on Sound 2 drops the size to ≈ 0.55 × (45 % smaller) while still decoding bit-perfectly. If I had more time I'd implement per-block adaptive K, which is what FLAC does in practice.