Exercise 2

Analysis

The application is an encoding and decoding app for uncompressed WAV files using the rice coding algorithm.

The encoding function `rice_encoder` takes a sample and a parameter `K` as input, where `K` represents the bit number, and calculates the quotient and remainder. The formula and methodology were adapted from the teachings provided in the Coursera Lab 9.005 Exercise 17. Implementing a rice encoder and decoder using Python, comprising of attaining the quotient and remainder, and generating a codeword for that sample.

The function `encoded_sample` uses the calculated remainder and bits to return the encoded sample in string.

The decoding function `rice_decoder` reverses the encoding process and reconstructs the original audio sample.

The `encode_and_decode` function combines the three aforementioned functions above. It reads the sound file in WAV format, encodes it using `rice_encoder` and `encoded_sample`, writes the encoded data to a new file with the extension "_Enc.ex2", then decodes the encoded file using `rice_decoder` and writes the decoded audio to a new WAV file with the extension "_EncDec.wav".

Additionally, the application at the end, prints the two sound files, 'Sound1.wav' and 'Sound2.wav', comparing the specifications and sizes of the original and the reconstructed audio files. The application demonstrates the implementation of rice coding, aiming to reduce or maintain the file size while preserving audio quality, as it is a lossless form of compression.

	Original Size	Rice (K=4bits)	Rice (K=2bits)	% Compression (K=4bits)	% Compression (K=2bits)
Sound1.wav	1MB	25MB	89MB	2400 %	8860 %
Sound2.wav	1MB	150MB	591MB	14900 %	59000 %

However, from the above table, we can infer that the size of the encoded file "_Enc.ex2", is much larger when K = 2 bits. The rice coding preserves the lossless data, but the compression actually produces a large encoded file. The reconstructed WAV file are exactly the same in terms of file size and quality though. The reason for the significant difference in file sizes between K=2 and K=4 is likely due to the trade-off between the length of the codes and the compression efficiency, as well as the fact we are using strings to store the unary codes, which takes up more bits. With K=2, shorter codes are used, leading to a larger file size, while with K=4, longer codes are used, resulting in a smaller encoded file size. In summary, when K is higher, the modulus attained is higher, resulting in longer and better generation of the code in unary and binary.

Further Development

We can improve the algorithm by reducing the encoded file size. This is done by utilizing bytearrays instead of strings, as bytearrays uses less storage compared to strings.