Exercise 2

The notebook uses the uploaded Exercise2_Files folder to work, although you may change this to your own directory and files if need be. It also stores the encoded and decoded files in this directory.

The application can be found here, the file named **Exercise 2.ipynb**:

https://hub.labs.coursera.org:443/connect/sharedojwzxqru?forceRefresh=false

The application reads a wav file as a byte object, each byte an integer from 0-255, and uses my implementation of the rice encoding algorithm to compress the file. It writes this file to an ex2 file as text, and uses that same file and a decoding algorithm to return the file to its original state.

After importing the necessary modules in the first cell, I define the rice encoding algorithm, based on a previous exercise. Using a unary helper function, the r_enc function takes an integer S and using K as a parameter, returns an encoded string of binary.

```
# unary function for use in encode
def unary(t):
    y=[];
    for i in range(t):
        y.append('1')
    y.append('0')
    return ''.join(y)

# rice encoding algorithm snippet
def r_enc(S, K):

    # find M
    M = 2 ** K

    # for S to be encoded, find:
    # quotient q = int(S/M)
    q = int(S//M)

    # remainder r = SmoduloM
    r = S%M

    # quotient code is q in unary
    q_un = unary(q)

    # remainder code is r in binary using K bits
    # func to return x in n-bit binary
    getbinary = lambda x, n: format(x, 'b').zfill(n)
    r_bin = getbinary(r, K)

    # codeword format <quotient code><remainder code>
    codeword = q_un + r_bin

    return codeword
```

In the next steps, we open the wav file, read it as a byte object and store it in the byte_data variable.

```
# getting binary data from wav file
w = wave.open('Exercise2_Files/Sound1.wav', 'rb')
byte_data = w.readframes(w.getnframes())
w.close()
```

Then, we compress each byte (integer from 0-255) using the rice encoder, specifying a K, and make a list of strings which I have called bits.

```
# bytes to bitstrings
bits = []
for byte in binary_data:
    bits.append(r_enc(byte, 4))
```

We then write the joined list of strings as text into the ex2 file as specified.

```
# encode and write to ex2 file
with open("Exercise2_Files/Sound1_Enc.ex2", "w") as f:
    # Writing data to a file
    f.write(' '.join(bits))
```

The r_dec function takes the codeword and a K value, and returns back the decoded integer.

```
# rice decoding algorithm snippet
def r_dec(codeword, K):
    cdw = list(codeword)

    # q by counting 1's before first 0
    q = 0

    for i in cdw:
        if i == '1':
            q += 1
        else:
            break

    # r reading next K bits as binary value
    r = ''.join(cdw[q:])
    r_int = int(r, 2)

    M = 2 ** K

    # S, encoded number, as q x M + r
    S = q * M + r_int

    return S
```

Reading our encoded file…

```
bitsagain = []

with open("Exercise2_Files/Sound1_Enc.ex2", "r") as fd:
    bitsagain = fd.read().split(' ')
```

And defining and using the decode_blocklist and r_dec functions, we get back our list of integers.

```
def decode_blocklist(blocklist, K):
    nums = []

    for block in blocklist:
        nums.append(r_dec(block, K))

    return nums
```

```
# decoded list of integers from bitstring
decbits = decode_blocklist(bitsagain, 4)
```

Converting those integers back to bytes gives us what should be our original byte object…

```
# conversion of int list to bytes
bytesagain = bytes(decbits)
```

We can check the equivalency of the first wav byte object (byte_data) and this one (bytesagain).

```
# equivalency check of files
bytesagain == byte_data
```

```
True
```

Then write it back to a wav file with the correct enc_dec extension.

```
with open('Exercise2_Files/Sound1_Enc_Dec.wav', mode='bx') as f:
    f.write(bytesagain)
```

All encoded files can be found in the Exercise2_Files folder if necessary.

The results of compression with K=4 and K=2 bits are in the following table:

	Original size	**Rice(K = 4 bits)**	**Rice(K = 2 bits)**	**% Compression (K = 4 bits)**	**% Compression (K = 2 bits)**
Sound1.wav	1 MB	13.1 MB	33.9 MB	-1210%	-3290%
Sound2.wav	1.01 MB	13.6 MB	35.8 MB	-1246.53%	-3444.55%

The resulting compressions took up more space, as the rice encoding algorithm uses more space to store binary values for larger numbers. It becomes especially inefficient for K=2. This can be easily seen from the results of a few calls to the r_enc function:

- r_enc(255, 4) = ‘1111111111111101111’

- r_enc(3, 4) = ‘00011’
- r_enc(244, 2) = ‘11111111111111111111111111111111111111111111111111111111111111000’
- r_enc(5, 2) = ‘1001’

Storing the top end of our byte values results in much larger file sizes. That’s the difference between a single byte (8 bit object) and a string of 64 bits in the case of K=2. Not so much compression going on here.

Further Development

Compressing a large binary string may be very difficult, as any one bit cannot tell us what the next one might be. Run-length encoding functions with a mix of integers and characters, and Huffman encoding works for more complex and varied data types like text. So instead, let us try to find the optimal K to match the 8 bit byte object with our largest integer value (255) with some runs of the r_enc function. We’re looking only for the length of the returned string.

- len(r_enc(255, 5)) = 13
- len(r_enc(255, 6)) = 10
- len(r_enc(255, 7)) = 9
- len(r_enc(255, 8)) = 9
- len(r_enc(255, 9)) = 10
- len(r_enc(255, 10)) = 11

We’ll have to stop at 10, as any larger K, according to the rice algorithm, we will have to use more and more bits to store the integer. We see an optimal solution at K = 7. Though this will also represent the lower integers (e.g. 1) using more bits, it will be more consistent. Let us try encoding the files with this K and updating the table.

	Original size	**Rice(K = 4 bits)**	**Rice(K = 2 bits)**	**Rice(K = 7 bits)**	**% Compression (K = 4 bits)**	**% Compression (K = 2 bits)**	**% Compression (K = 7 bits)**
Sound1.wav	1 MB	13.1 MB	33.9 MB	9.49 MB	-1210%	-3290%	-849%
Sound2.wav	1.01 MB	13.6 MB	35.8 MB	9.58 MB	-1246.53%	-3444.55%	-770.9%

Still exceeding the original file by quite the amount, though an improvement over our original encodings with K = 4 and 2.