ELL786 REPORT

ASSIGNMENT 2

SARANSH AGARWAL (2019MT60763)
HARSH SHARMA (2019MT60628)

Experiment 1:

Implement the LZW encoder

```
# encoding function takes string and initial dictionary as input and encodes the string
# d0 is the initial dictionary primed with the alphabet set
# the alphabet set keys must start from 1 to its length with increments of 1

def encoding(s, d0):

    code_table = d0
    n = len(s)
    p = ''

    code_stream = []
    k = len(d0)+1

    for i in range(n):
        c = s[i]

        if p + c not in code_table:
            code_table[p + c] = k
            code_stream.append(code_table[p])
            p = c
            k += 1
        else:
            p += c

    #printing dictionary
    #print(code_table)
    code_stream.append(code_table[p])
    return code_stream
```

Figure 1: Wrapper Code

```
# decoding function takes codes and initial dictionary as input and decodes the string

def decoding(v, d0):

    # invert d0 mapping
    Dict = {v: k for k, v in d0.items()}

    k = ""
    old = v[0]
    s = Dict[old]
    c = ""
    c += s[0]
    k += s

    count = len(d0)+1

    for i in range(len(v) - 1):
        n = v[i + 1]

        if n not in Dict:
            s = Dict[old]
            s = s + c
        else:
            s = Dict[n]
        k += s
        c = ""
        c += s[0]
        Dict[count] = Dict[old] + c
        count += 1
        old = n

    return k
```

Figure 2: Wrapper Code

Testing against an initial dictionary consisting of the letters a, b, r, y, ·, then encode the **"a· bar· array· by· barrayar· bay"** message using the LZW algorithm.

INDEX	ENTRY
1	a
2	b
3	r
4	y
5	·

Figure 3: Initial Dictionary

INDEX	ENTRY
1	a
2	b
3	r
4	y
5	·
6	a·
7	·b
8	ba
9	ar
10	r·
11	·a
12	arr
13	ra
14	ay
15	y·
16	·by
17	y·b
18	bar
19	rr
20	ray
21	ya
22	ar·
23	·ba

Figure 4: LZW dictionary for encoding message

The encoded sequence for the message **"a· bar· array· by· barrayar· bay"** is

1,5,2,1,3,5,9,3,1,4,7,15,8,3,13,4,9,7,14

Now, convert the bmp file to a binary string

```
#Reading image form bmp format
image = Image.open("sample.bmp")
img = np.array(image).flatten()

#converting the bmp format image into binary string
m = ''.join(['{0:08b}'.format(num) for num in img])
print(len(m))
```

Figure 5: convert the bmp file to a binary string

```
#initial dictionary containes only 0 and 1
d = {"0": 1,"1": 2}

#code is the generated codes by encoding
code = encoding(m, d)

#finding compression ratio

#converting codes into binary format
#print(max(code))
p = ''.join(['{0:020b}'.format(num) for num in code])
print(len(p))


print("compression_ratio",(len(m))/len(p))
```

Figure 6: Wrapper Code

CONCLUSION:

Length of binary string before compression = 39321600 bits

Length of binary string after compression = 13359700 bits

Compression ratio = (39321600)/ (13359700) = 2.94

Encoding scheme	Length of binary string before compression(bits)	Length of binary string after compression(bits)	Compression Ratio
LZW	39321600	13359700	2.94

Figure 7: compression ratio for LZW

Experiment 2:

Implement the GIF encoder

```
def encoding(s, d0):

    code_table = d0
    n = len(s)
    p = ''

    s1=""
    k = len(d0)+1
    for i in range(n):
        c = s[i]

        if p + c not in code_table:
            code_table[p + c] = k

            # increasing pointer length by one as the dictionary size doubles
            if(k<512):
                s1+=('{0:09b}'.format(code_table[p]))
            elif(k>=512  and k<1024):
                s1+=('{0:010b}'.format(code_table[p]))
            elif(k>=1024 and k<2048):
                s1+=('{0:011b}'.format(code_table[p]))
            elif(k>=2048 and k<4096):
                s1+=('{0:012b}'.format(code_table[p]))


            p = c
            k += 1

            # when dictionary size gets equal to 4096 then discard the prev dictionary
            # and start a new dictionary

            if(k == 4096):
                Dict = dict()
                for i in range(256):
                    t = ""
                    t += chr(i)
                    Dict[t] = i+1
                Dict['clear_code'] = 257
                Dict['EOF'] = 258
                code_table = Dict
                s1 += (bin(257))[2:]
                k = 259
                p = ''


        else:
            p += c


    if(k<512):
        s1+=('{0:09b}'.format(code_table[p]))
    elif(k>=512 and k<1024):
        s1+=('{0:010b}'.format(code_table[p]))
    elif(k>=1024 and k<2048):
        s1+=('{0:011b}'.format(code_table[p]))
    elif(k>=2048 and k<4096):
        s1+=('{0:012b}'.format(code_table[p]))

    s1+=(bin(258))[2:]
    return s1
```

Figure 8: Wrapper Code

Now, convert the bmp file to a binary string

```
#Reading image form bmp format
image = Image.open("sample.bmp")
img = np.array(image).flatten()

#converting the bmp format image into binary string
m = ''.join(['{0:08b}'.format(num) for num in img])
print(len(m))
```

Figure 9: convert the bmp file to a binary string

```
#code is the generated codes by encoding
code = encoding(m, Dict)
print(len(code))

#finding compression ratio
print("compression_ratio",(len(m))/len(code))
```

Figure 10: Wrapper Code

COMPRESSION RATIO:

Length of binary string before compression = 39321600 bits

Length of binary string after compression = 20321360 bits

Compression ratio = (39321600)/ (20321360) = 1.94

COMPARISION BETWEEN LZW AND GIF ENCODER

Encoding scheme	Length of binary string before compression(bits)	Length of binary string after compression(bits)	Compression Ratio
LZW	39321600	13359700	2.94
GIF	39321600	20321360	1.94

Figure 11: compression ratio for LZW and GIF

CONCLUSION:

From figure 13 we can conclude that compression ratio by LZW technique is 2.94 while the compression ratio by GIF encoder is 1.94. The compression ratio for GIF is less than LZW. In GIF we use two additional special codes clear codes and End of Information codes which is not there in LZW. In GIF the pointer gets incremented by one from dictionary to dictionary due to which GIF has variable compression sizes. In GIF the dictionary becomes static as the size of dictionary becomes equal to 4096. Due to which in GIF the computational complexity increases. That's why we are getting less compression ratio for GIF than LZW.

Additional techniques used in gif:

The Graphics Interchange Format (GIF) was developed by CompuServe Information Service to encode graphical images. It is another implementation of the LZW algorithm and is very similar to the compress command.GIF uses dynamic dictionary for compression.

Algorithm:

- It takes number of bits per pixel b as parameter.
- It uses initial dictionary of size 2^(b+1). When this dictionary fills up, we doubled the dictionary size until we reach maximum dictionary size of 4096.At this point dictionary behaves as static dictionary.
- The codewords from the LZW algorithm are stored in blocks of characters. Pointer get longer one byte from dictionary to dictionary and output are in block of 8 bytes.
- After reaching maximum size of dictionary 4096 we discard the old dictionary and start new dictionary.
- At the time of starting new dictionary, Encoder emits **clear code** which is sign for decoder to discard the dictionary.
- The end of compressed file is denoted by an **END OF INFORMATION** code.

Experiment 3:

Implement the PNG encoder

```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import zlib
from functools import reduce

# Reading image form bmp format
image = Image.open("sample.bmp")
img = np.array(image).flatten()

# converting the bmp format image into binary string
m = ''.join(['{0:08b}'.format(num) for num in img])
print(len(m))

# compression using png encoder
new_string = bytearray(m, "ascii")
p = zlib.compress(new_string, level=9)

p_len = reduce(lambda x, y: x+len(bin(y)[2:]), p, 0)
print(p_len)

print("compression_ratio", len(m) / p_len)
```

Figure 12: Wrapper Code

Now, convert the bmp file to a binary string

```
#Reading image form bmp format
image = Image.open("sample.bmp")
img = np.array(image).flatten()

#converting the bmp format image into binary string
m = ''.join(['{0:08b}'.format(num) for num in img])
print(len(m))
```

Figure 13: convert the bmp file to a binary string

COMPRESSION RATIO:

Length of binary string before compression = 39321600 bits

Length of binary string after compression = 10086074 bits

Compression ratio = (39321600)/ (10086074) = 3.89

COMPARISION BETWEEN LZW AND PNG ENCODER

Encoding scheme	Length of binary string before compression(bits)	Length of binary string after compression(bits)	Compression Ratio
LZW	39321600	13359700	2.94
PNG	39321600	10086074	3.89

Figure 14: compression ratio for LZW and PNG

CONCLUSION:

From figure 17 we can conclude that compression ratio by LZW technique is 2.94 while the compression ratio by PNG encoder is 3.89. The compression ratio for PNG is more than LZW. For lossless compression, a better method is to predict pixels based on neighbouring pixels already seen, then compress the residual prediction error. PNG uses this format, using deflate to compress the residual. That's why we are getting more compression ratio for PNG than LZW.

Additional techniques used in PNG:

The PNG standard is one of the first standards to be collaboratively developed over the Internet. The impetus for it was an announcement in December 1994 by Unisys (which had acquired the patent for LZW from Sperry) and CompuServe that they would start charging royalties to authors of software that included support for GIF.

- The PNG encoder uses deflate implementation of LZ77 technique.
- In PNG encoder the match length is between 2 and 258.
- In PNG technique at each step three bytes of data is examined. If we can't find a match of at least three bytes then we put first byte and examined next three bytes.
- At each step we put value of single byte or the pair <match length, offset >.
- The total alphabet size is 286 The indices 0-255 represent literal bytes and the index 256 is an end-of-block symbol. The remaining 29 indices represent codes for ranges of lengths between 3 and 258.
- The index values are represented using a Huffman code.
- The offset takes values between 1 and 32678.These values are divided into 30 ranges and these 30 range values are encoded using Huffman codes.

Index	# of selector bits	Length	Index	# of selector bits	Length	Index	# of selector bits	Length
257	0	3	267	1	15,16	277	4	67–82
258	0	4	268	1	17,18	278	4	83–98
259	0	5	269	2	19–22	279	4	99–114
260	0	6	270	2	23–26	280	4	115–130
261	0	7	271	2	27–30	281	5	131–162
262	0	8	272	2	31–34	282	5	163–194
263	0	9	273	3	35–42	283	5	195–226
264	0	10	274	3	43–50	284	5	227–257
265	1	11, 12	275	3	51–58	285	0	258
266	1	13, 14	276	3	59–66			

Figure 15: code representation for match length 59.

Index Ranges	# of bits	Binary Codes
0–143	8	00110000 through 10111111
144–255	9	110010000 through 111111111
256–279	7	0000000 through 0010111
280–287	8	11000000 through 11000111

Figure 16: Huffman codes for match length 59.

COMPARISION BETWEEN LZW, GIF AND PNG ENCODER

Encoding scheme	Length of binary string before compression(bits)	Length of binary string after compression(bits)	Compression Ratio
LZW	39321600	13359700	2.94
GIF	39321600	20321360	1.94
PNG	39321600	10086074	3.89

Figure 17: compression ratio for LZW, GIF and PNG

CONCLUSION:

From figure 17 we can conclude that compression ratio by LZW technique is 2.94 while the compression ratio by GIF encoder is 1.94 and compression ratio by PNG encoder is 3.89.

The compression ratio for GIF is less than LZW and PNG. GIF uses variant of LZW. In GIF we use two additional special codes clear codes and End of Information codes which is not there in LZW. In GIF the pointer gets incremented by one from dictionary to dictionary due to which GIF has variable compression sizes. In GIF the dictionary becomes static as the size of dictionary becomes equal to 4096. Due to which in GIF the computational complexity increases. That's why we are getting less compression ratio for GIF than LZW and PNG.

The compression ratio for PNG is more than LZW and GIF.PNG encoder uses deflate compression technique. For lossless compression, a better method is to predict pixels based on neighbouring pixels already seen, then compress the residual prediction error. PNG uses this format, using deflate to compress the residual. That's why we are getting more compression ratio for PNG than LZW and GIF.

From the above we conclude PNG is best encoder for compressing an image while GIF is the worst among LZW, GIF and PNG.

Experiment 4:

USING JPEGLS ENCODER

JPEG-LS (1998) is a lossless image format created as a replacement for the lossless JPEG mode, which is inefficient. This compression method consists of a predictor and a subsequent context entropic coder (Golomb-Rice coder). Its implementation is provided in python in the following code snippets and the attached zip file which contains actual source code.

```
import math


def int_binary(r, k):
    res = ''
    while k != 0:
        res = str(r % 2) + res
        r = r // 2
        k -= 1
    return res


def unary_code(q):
    # unary code of q
    return '0' * q + '1'


def codeword_remainder(r, m):
    if 0 <= r <= int(math.pow(2, math.ceil(math.log2(m)))) - m - 1:
        return int_binary(r, int(math.floor(math.log2(m))))
    else:
        return int_binary(r + int(math.pow(2, math.ceil(math.log2(m)))) - m, math.ceil(math.log2(m)))


def golomb_encode(m, n):
    q = n // m
    r = n % m
    return unary_code(q) + codeword_remainder(r, m)


def golomb_power_of_two(y, k):
    return unary_code(y >> k) + int_binary(y & ((1 << k) - 1), k)

# b = math.ceil(log2(M)) , where M is the alphabet-set size
# L - maximum codeword length
# constraint: L > b+1
def modified_GPO2(y, k, b, L):
     q = y >> k
     q_max = L - b - 1
     if q < q_max:
          return golomb_power_of_two(y, k)
     else:
          return unary_code(q_max) + int_binary(y - 1, b)


def modified_GPO2_decode(s, b, L):
     q_max = L - b - 1
     if s[:q_max] == '0' * q_max:
          # escape-code identified
          return int(s[q_max + 1:], 2) + 1
     else:
          # simply encode y using GPO2
          i = 0
          while s[i] != '1':
               i += 1
          return int(bin(i)[2:] + s[i+1:], 2)
```

Figure 18: Wrapper Code

```python
import numpy as np

from golomb_code import modified_GPO2, modified_GPO2_decode
import math


# return the grayscale value of pixel at i,j
# 0 if pixel doesn't exist
def get(grayscale_img, i, j, m, n):
    return grayscale_img[i][j] if 0 <= i < m and 0 <= j < n else 0


# quantize local gradient bases on user defined parameter t1 ,t2 ,t3
def def_component(d, t1, t2, t3):
    if d <= -t3:
        return -4
    elif -t3 < d <= -t2:
        return -3
    elif -t2 < d <= -t1:
        return -2
    elif -t1 < d <= 0:
        return -1
    elif d == 0:
        return 0
    elif 0 < d <= t1:
        return 1
    elif t1 < d <= t2:
        return 2
    elif t1 < d <= t3:
        return 3
    else:
        # t3 < d
        return 4


def context_map(c1, c2, c3):
    # assuming (c1, c2, c3) is normalized (i.e. first non-zero entry is positive)
    if c1 != 0:
        return 41 + (c1 - 1) * 81 + (c2 + 4) * 9 + (c3 + 5)
    elif c2 != 0:
        # c1 == 0
        return 5 + (c2 - 1) * 9 + (c3 + 5)
    else:
        # c1 == 0 and c2 == 0
        return c3 + 1


# define context vector given local gradient (d1, d2, d3) and user defined parameter values (for quantization)
def def_context(d1, d2, d3, t1=3, t2=7, t3=21):
    return [def_component(d1, t1, t2, t3), def_component(d2, t1, t2, t3), def_component(d3, t1, t2, t3)]


# maps the residue res for golomb coding
def map_residue(res, k, B, N):
    if k == 0 and 2 * B <= -N:
        if res >= 0:
            return 2 * res + 1
        else:
            return -2 * (res + 1)
    else:
        if res >= 0:
            return 2 * res
        else:
            return -2 * res - 1
```

Figure 19: Wrapper Code

```
def invmap_residue(mapped_res, k, B, N):
    if k == 0 and 2 * B <= -N:
        if mapped_res % 2 == 0:
            return (-mapped_res // 2) - 1
        else:
            return (mapped_res - 1) // 2
    else:
        if mapped_res % 2 == 0:
            return mapped_res // 2
        else:
            return - (mapped_res + 1) // 2


# de-correlation + refinement step
# returns residual sequence of refined prediction
def jpegls_encode(grayscale_img, M, N0=64):
    # height
    m = len(grayscale_img)

    # width
    n = len(grayscale_img[0])

    # define cum_count, cum_bias,cum_ares for a given context {0, ..., 364}
    cum_count = [1] * 365
    cum_bias = [0] * 365
    correction = [0] * 365
    cum_ares = [max(2, (M + 32) // 64)] * 365

    beta = math.ceil(math.log2(M))
    bbeta = max(2, math.ceil(math.log2(M)))
    # any positive integer > bbeta+1 works
    L_max = 2 * (bbeta + max(8, bbeta))

    residual_seq = np.zeros((m, n), dtype=object)
    for i in range(m):
        for j in range(n):
            # de-correlation via prediction
            a = get(grayscale_img, i, j - 1, m, n)
            b = get(grayscale_img, i - 1, j, m, n)
            c = get(grayscale_img, i - 1, j - 1, m, n)
            d = get(grayscale_img, i - 1, j + 1, m, n)

            # context modeling (refining prediction error)

            # local gradient
            g1 = int(d) - int(b)
            g2 = int(b) - int(c)
            g3 = int(c) - int(a)

            # using default values of quantization boundaries to define context
            c_vec = def_context(g1, g2, g3)

            # 'normalize' c such that first non-zero entry of c is positive
            SIGN = 1
            if c_vec[0] < 0:
                c_vec = [-x for x in c_vec]
                SIGN = -1
            elif c_vec[0] == 0 and c_vec[1] < 0:
                c_vec = [-x for x in c_vec]
                SIGN = -1
            elif c_vec[0] == 0 and c_vec[1] == 0 and c_vec[2] < 0:
                c_vec = [-x for x in c_vec]
                SIGN = -1

            # value-given the context vector c
            context = context_map(*c_vec) - 1

            # fixed prediction
            predicted_val = int(min(a, b)) if c >= max(a, b) else int(max(a, b)) if c <= min(a, b) else int(a) + int(
                b) - int(c)
```

Figure 20: Wrapper Code

```
        predicted_val = predicted_val + SIGN * correction[context]

        if predicted_val >= M:
            predicted_val = M - 1
        elif predicted_val < 0:
            predicted_val = 0

        residue = SIGN * (int(grayscale_img[i][j]) - predicted_val)

        # "modulo" the residue so that it belongs to -M/2 to M/2
        if residue < -M // 2:
            residue = residue + M
        elif residue > M // 2:
            residue = residue - M

        # encode residual_seq[i, j] using Golomb-Codes
        # compute the context dependent golomb-parameter k
        k = 0
        while (cum_count[context] << k) < cum_ares[context]:
            k += 1

        # perform mapped error encoding
        mapped_res = map_residue(residue, k, cum_bias[context], cum_count[context])
        residual_seq[i, j] = modified_GPO2(mapped_res, k, beta, L_max)

        # update cum_count , cum_bias given context c
        cum_bias[context] += residue
        cum_ares[context] += abs(residue)

        # if reset threshold limit is reached
        if cum_count[context] == N0:
            cum_count[context] = cum_count[context] >> 1
            cum_bias[context] = cum_bias[context] >> 1 if cum_bias[context] >= 0 else -(
                    (1 - cum_bias[context]) >> 1)
            cum_ares[context] = cum_ares[context] >> 1
             cum_count[context] += 1

             # division-free bias computation
             if cum_bias[context] <= -cum_count[context]:
                 correction[context] = correction[context] - 1
                 cum_bias[context] += cum_count[context]
                 if cum_bias[context] <= -cum_count[context]:
                     cum_bias[context] = -cum_count[context] + 1

             elif cum_bias[context] > 0:
                 correction[context] = correction[context] + 1
                 cum_bias[context] -= cum_count[context]
                 if cum_bias[context] > 0:
                     cum_bias[context] = 0
    return residual_seq


def jpegls_decode(residual_seq, M, N0=64):
    # height
    m = len(residual_seq)

    # width
    n = len(residual_seq[0])

    # define cum_count, cum_bias,cum_ares for a given context {0, ..., 364}
    cum_count = [1] * 365
    cum_bias = [0] * 365
    correction = [0] * 365
    cum_ares = [max(2, (M + 32) // 64)] * 365

    beta = math.ceil(math.log2(M))
    bbeta = max(2, math.ceil(math.log2(M)))
    # any positive integer > bbeta+1 works
    L_max = 2 * (bbeta + max(8, bbeta))
```

Figure 21: Wrapper Code

```
grayscale_img = np.zeros((m, n), dtype=int)
for i in range(m):
    for j in range(n):
        # de-correlation via prediction
        a = get(grayscale_img, i, j - 1, m, n)
        b = get(grayscale_img, i - 1, j, m, n)
        c = get(grayscale_img, i - 1, j - 1, m, n)
        d = get(grayscale_img, i - 1, j + 1, m, n)

        # context modeling (refining prediction)

        # local gradient
        g1 = int(d) - int(b)
        g2 = int(b) - int(c)
        g3 = int(c) - int(a)

        # using default values of quantization boundaries to define context
        c_vec = def_context(g1, g2, g3)

        # 'normalize' c such that first non-zero entry of c is positive
        SIGN = 1
        if c_vec[0] < 0:
            c_vec = [-x for x in c_vec]
            SIGN = -1
        elif c_vec[0] == 0 and c_vec[1] < 0:
            c_vec = [-x for x in c_vec]
            SIGN = -1
        elif c_vec[0] == 0 and c_vec[1] == 0 and c_vec[2] < 0:
            c_vec = [-x for x in c_vec]
            SIGN = -1

        # value-given the context vector c
        context = context_map(*c_vec) - 1

              cum_count[context] = cum_count[context] >> 1
              cum_bias[context] = cum_bias[context] >> 1 if cum_bias[context] >= 0 else -(
                      (1 - cum_bias[context]) >> 1)
              cum_ares[context] = cum_ares[context] >> 1
          cum_count[context] += 1

          # above residue is "modulo-'ed" in [-M/2,M/2]
          # re-map to get actual "corrected" residue
          if -predicted_val <= residue < M - predicted_val:
              residue = residue
          elif residue >= M - predicted_val:
              residue = residue - M
          elif residue < -predicted_val:
              residue = residue + M
          residue = SIGN * residue

          grayscale_img[i, j] = (residue + predicted_val) % M

          # Implement the 15th - pt


          # division-free bias computation
          if cum_bias[context] <= -cum_count[context]:
              correction[context] = correction[context] - 1
              cum_bias[context] += cum_count[context]
              if cum_bias[context] <= -cum_count[context]:
                  cum_bias[context] = -cum_count[context] + 1

          elif cum_bias[context] > 0:
              correction[context] = correction[context] + 1
              cum_bias[context] -= cum_count[context]
              if cum_bias[context] > 0:
                  cum_bias[context] = 0

    return grayscale_img
```

Figure 22: Wrapper Code

The JPEG-LS/LOCO-I Lossless Image Compression Algorithm

Read Image and store it's RGB values

In [1]:
```
import jpeg_ls
import imageio as iio
from functools import reduce
import numpy as np
import math
from IPython.display import Image, display
import zlib

# img is in RGBA format , opacity 0 for png images
img = iio.imread("./images/earth.png", format='png')

# get grayscale values for each color
red_img = [[pixel[0] for pixel in row] for row in img]
green_img = [[pixel[1] for pixel in row] for row in img]
blue_img = [[pixel[2] for pixel in row] for row in img]

width = len(red_img[0])
height = len(red_img)

M = 256  # alphabet set
```

Encode each monochromatic grayscale part

In [2]:
```
# JPEG-LS encoding

encode_red_jpeg = jpeg_ls.jpegls_encode(red_img, M)
encode_green_jpeg = jpeg_ls.jpegls_encode(green_img, M)
encode_blue_jpeg = jpeg_ls.jpegls_encode(blue_img, M)

# PNG encoding

encode_red_png = [zlib.compress(bytearray(row), level=-1) for row in red_img]
encode_green_png = [zlib.compress(bytearray(row), level=-1) for row in green_img]
encode_blue_png = [zlib.compress(bytearray(row), level=-1) for row in blue_img]
```

Compare the size before and after compression for:

1. RGB Image

2. JPEG Encoded File

3. PNG Encoded File

In [3]:
```
len_red_jpeg = reduce(lambda x, y: x + reduce(lambda a, b: a + len(b), y, 0), encode_red_jpeg, 0)
len_green_jpeg = reduce(lambda x, y: x + reduce(lambda a, b: a + len(b), y, 0), encode_green_jpeg, 0)
len_blue_jpeg = reduce(lambda x, y: x + reduce(lambda a, b: a + len(b), y, 0), encode_blue_jpeg, 0)

len_red_png = reduce(lambda x, y: x + reduce(lambda a, b: a + len(bin(b)[2:]), y, 0), encode_red_png, 0)
len_green_png = reduce(lambda x, y: x + reduce(lambda a, b: a + len(bin(b)[2:]), y, 0), encode_red_png, 0)
len_blue_png = reduce(lambda x, y: x + reduce(lambda a, b: a + len(bin(b)[2:]), y, 0), encode_red_png, 0)


# Original size of image
# 8-bits/pixel color
print("original length", 3 * 8 * width * height // 1024, 'kB')

# Size of image after compressing (via JPEG-LS)
print("compressed length in jpeg", (len_red_jpeg + len_green_jpeg + len_blue_jpeg) // 1024, 'kB')

# size of image after compressing (via PNG encoded file)
print("compressed length in png", (len_red_png + len_green_png + len_blue_png) // 1024, 'kB')
```

```
original length 13824 kB
compressed length in jpeg 5120 kB
compressed length in png 6137 kB
```

Figure 23: Wrapper Code

Decompress each encoded file

In [4]:
```
# decompression step (JPEG-LS)

decoded_red_jpeg = jpeg_ls.jpegls_decode(encode_red_jpeg, M)
decoded_green_jpeg = jpeg_ls.jpegls_decode(encode_green_jpeg, M)
decoded_blue_jpeg = jpeg_ls.jpegls_decode(encode_blue_jpeg, M)

image_matrix_jpeg = [[(r, g, b) for r, g, b in zip(row_red, row_green, row_blue)]
                for row_red, row_green, row_blue in zip(decoded_red_jpeg, decoded_green_jpeg, decoded_blue_jpeg


# decompression step (PNG)

decoded_red_png = [zlib.decompress(row) for row in encode_red_png]
decoded_green_png = [zlib.decompress(row) for row in encode_green_png]
decoded_blue_png = [zlib.decompress(row) for row in encode_blue_png]

image_matrix_png = [[(r, g, b) for r, g, b in zip(row_red, row_green, row_blue)]
                    for row_red, row_green, row_blue in zip(decoded_red_png, decoded_green_png, decoded_blue_pn


# Write new image onto the disk

newimg_jpegls = iio.imwrite('./images/earth_compressedjpeg.jpeg', np.uint8(image_matrix_jpeg))
newimg_png = iio.imwrite('./images/earth_compressedpng.png', np.uint8(image_matrix_png))
```

Comparison of original vs compressed image

In [5]:
```
display(Image(filename="./images/earth.png"))
display(Image(filename="./images/earth_compressedjpeg.jpeg"))
display(Image(filename="./images/earth_compressedpng.png"))
```

Figure 24: Wrapper Code

Original Image

JPEG-LS Compressed Image

PNG Compressed Image

Figure 25: Original vs JPEG-LS vs PNG

CONCLUSIONS

Similar to how we compressed earth image, to compare PNG and JPEG-LS we compressed multiple images and summarize their result in the following table

Image	Original Size	JPEG-LS	PNG	CR(JPEG-LS)	CR(PNG)
Lena	90037 KB	33829 KB	69337 KB	2.66	1.30
Earth	13824 KB	5120 KB	6137 KB	2.70	2.25
Sea	22443 KB	13788 KB	16061 KB	1.62	1.39
Nature	7763 KB	6146 KB	6444 KB	1.26	2.20
Facebook logo	16911 KB	3819 KB	3527 KB	4.43	4.80
Coffee	1780 KB	871 KB	649 KB	2.04	2.74
Spiderman	3037 KB	1252 KB	1095 KB	2.42	2.77

Figure 26: Compressed image size for different images

These images (original, jpeg compressed, png compressed) can be found in the images directory inside assignment zip folder.

We can observe that, for more natural palletized images, JPEG-LS seems to do better than PNG.

For dithered, halftoned, and some graphic images for which LZ-type methods are better suited PNG does better than JPEG-LS

PSNR (Peak – Signal – To – Noise – Ratio):

Computing the Peak-Signal-To-Noise-Ratio (PSNR) Values

PSNR is most easily defined via the mean squared error (MSE). Given a noise-free m×n monochrome image I and its noisy approximation K, MSE is defined as

$MSE = \frac{1}{m\,n}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2$. The PSNR (in dB) is defined as

$$\begin{aligned} PSNR &= 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right) \\ &= 20 \cdot \log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right) \\ &= 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE). \end{aligned}$$

Here, MAXI is the maximum possible pixel value of the image. When the pixels are represented using 8 bits per sample, this is 255. More generally, when samples are represented using linear PCM with B bits per sample, MAXI is 2B − 1.

Application in color images

For color images with three RGB values per pixel, the definition of PSNR is the same except that the MSE is the sum over all squared value differences (now for each color, i.e. three times as many differences as in a monochrome image) divided by image size and by three.

```
# PSNR value for LPEG-LS

MSE = 0.0
size = width * height
for i in range(height):
    for j in range(width):
        MSE += ((red_img[i][j] - decoded_red[i][j]) ** 2)

for i in range(height):
    for j in range(width):
        MSE += ((green_img[i][j] - decoded_green[i][j]) ** 2)

for i in range(height):
    for j in range(width):
        MSE += ((blue_img[i][j] - decoded_blue[i][j]) ** 2)

MSE = MSE / (3 * size)
print(MSE)

# Since JPEG-LS/LOCO-I is lossless , we expect it to be 0.0 which is the case

# print("PSNR for JPEG-LS ", 20 * math.log10(255/math.sqrt(MSE)))
```

Figure 27: Wrapper Code

CONCLUSIONS

PSNR (Peak Signal to Noise Ratio) for JPEG-LS and PNG compressed images is not defined as these techniques are lossless compression techniques and hence observed **MSE** (Mean Squared Error) in both JPEG-LS and PNG is 0.0 (which is indicated in the output of Fig. 27 code snippet).