Image Representation with Lossy and Lossless Encoding Techniques

Team Members:

Casey Kuball

Glenn Craver

Steve Carr

Wesley Rose

Dawne Flanagan

**Abstract**:  
 Images today can be stored in several different types of digital representations. This process of changing the representation of an image is called image coding and if the resulting image uses less storage space than the original then it is called image compression. This project was designed to breakdown an image representation into it’s three separate color channels, quantize those color channels using a specified color space then compress and encode each of the color channels, independently from one another, into a single binary file. Using the Python Imaging Library API and the documentation provided to the group, the program was implemented in Python and performs all of the necessary functions as dictated in the assignment’s requirements. By completing this project, one can directly observe the effects of compressing an image by first reducing it’s individual color channels then encoding each color channel back into the original picture representation.   
  
  
**Keywords**:

Python 2.7.3  
 Python Imaging Library (PIL)  
 RGB

YCbCr

Color model

Color space

Quantization

Predictive Coding

Predictor

Encoding Schemes

Variable-Length encoding

Shannon-Fano coding

Dictionary encoding

LZW coding

Signal-to-Noise ratio

**Introduction**

Terminology

Throughout this report, different color models will be referenced using their acronym, which traditionally describes the major components in the color model. Various methods of quantization (i.e., color reduction) will be discussed, as well as different models of compression techniques (e.g., Variable-length encoding with Shannon-Fano coding). When the term ‘API’ is referenced, it is used to denote Application Programming Interface. The only API used in this project is the Python Imaging Library.

Goal

The goal of this project is to implement a program which utilizes a 3rd party imaging API and implements the following tasks when supplied a color image:

* + Reduce the resolution of the three color channels in the image
  + Implement quantization schemes with the following options:
    - No quantization
    - Uniform quantization of the color spaces into three bins
  + Implement eight predictive coding schemes where each schema has a unique predictor on the quantized data; predictors listed in appendix A3.
  + Implement error quantization with the following options:
    - No quantization
    - Quantization of the error into a user defined set of bins
  + Implement encoding schemes with the following options:
    - No encoding
    - Variable length with Shannon-Fano coding
    - Dictionary encoding with LZW coding
  + Output the a binary file with extension .RGB or .YBR and the signal-to-noise ratio.

Assumptions

The program assumes that the user is aware of the functions that are

implemented and has previous knowledge of the various color models as well as the mechanics of image quantization and compression.

**Proposed Solution**

The proposed solution to the goals and requirements was to implement a program utilizing the Python Imaging Library. Given an image provided by the user, the program will perform the functions described in the ‘Goal’ portion above and produce a binary file with extension .RGB or .YBR as output. The program is run locally and the pathname to the image is supplied by the user at runtime, as well as any additional arguments that are needed (e.g., Amount of resolution reduction to perform).

When initializing the program it will ask the user to provide an image to work with. After providing the file path to an image the user will be asked to enter six integers which determines how much the dimensions of each color channel will be reduced by; resolution reduction algorithms listed in appendix A1. After reducing the channels they are then passed to the quantization task.

The quantization schemes are broken into two categories:

* Uniform: The color space is broken into three bins where the number of bins, N1, N2 and N3 are less than or equal to K, where K = 256.
* Non-Uniform: No quantization.

When performing the quantization the user will be prompted to enter no quantization or uniform quantization, in which if uniform quantization is selected they will supply N1, N2 and N3. The color space of each color channel is then divided by the user provided N value and each of the colors is mapped to a region which it falls in; algorithms listed in appendix A2. The colors for each region are now the average of all the colors mapped to that region. After quantizing each color channel they are then passed to the predictive coding task.

The predictive coding schemas consist of eight different predictors all of which have different impacts on the quantized data. The predictors work by taking the individual quantized color channels from the previous task, the width and height of the image, and computing the prediction algorithm the user selects when the predictive coding task started. It then zips the color channels back together and returns a list of error values that were calculated during the prediction algorithm computation; algorithms listed in appendix A3. This list of error values are then passed to the error quantization schemes.

After the predictive coder finishes the error quantization schemes will either perform uniform or non-uniform quantization of the error into m bins, where m is supplied by the user. The error quantization works in a similar fashion as the quantization schemes used in task two, but differs in that the number of bins is not limited to three so the new list of quantized errors will look more uniform; algorithms listed in appendix A4.

The encoding schemes that are applied to the error quantization data are implemented using similar techniques described in the book “*Fundamentals of Multimedia”* as well as from resources around the internet, such as “Project: What's In A GIF - LZW Image Data”. The Shannon-Fano implementation works as follows:

1. Given an image, the algorithm evaluates the frequency of the colors found in the image and saves them to a list.
2. It then sort’s the list according to frequency in descending order using quicksort.
3. Then the left part of the list is assigned a 0 and the right part is assigned a 1
4. Steps 2 and 3 are repeated until each color frequency has a corresponding code.

Shannon-Fano algorithm listed in appendix A4 and LZW algorithm listed in appendix A5..

The viewer that displays the binary file was a predefined function, image.show(), from the PIL API. However, PIL's image.show() can have problems on windows, as some viewers do not work properly.

**Interface Specifications**

The interface for this project is accessed through the command prompt or terminal. The interface drives the program and acts as a messenger which collects input from the user during each of the aforementioned ‘Goal’ tasks then passes that information to each respective function. The functions performing a tasks functionality are stored in separate class file’s in which all of the computation behind that specific task is performed by the methods in that class file then returned to the interface class. The information is passed in a linear fashion where task one supplies task two and task two supplies task three and so on until all tasks are completed.

The use of the Python Imaging Library API was extremely useful when implementing our proposed solution to the tasks listed in the Goal section. The PIL API contains over a dozen modules each containing a considerable amount of functions to help achieve a successful implementation. One example is the use of the function getdata() which is called on an Image. Using this built in function made it extremely easy to retrieve the contents of an image as a sequence object containing the image pixel values.

**System Requirements**

The system requirements are as follows:

* + Python 2.7.3 (This is the latest stable release)
  + Python Imaging Library 1.1.7
  + Mac OS X 10.6+, Windows XP or Later (any OS with Python is suitable but installation instructions are only given for Windows)
  + Root permission level to change Path variables

**Installation Instructions**

1. Download and install the x86 (32-bit) version of python 2.7 (<http://www.python.org/download/>).
2. Add the python installation directory (e.g. C:\Python27) to your $PATH

environmental variable.

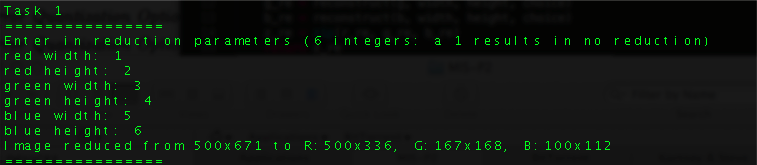
1. Download and install the latest version of PIL for Python 2.7 (<http://www.pythonware.com/products/pil/>).

**Execution Instructions**

1. Open up a DOS or Terminal prompt and navigate to the folder where the program is located
2. Type ‘python main.py’ to run the program



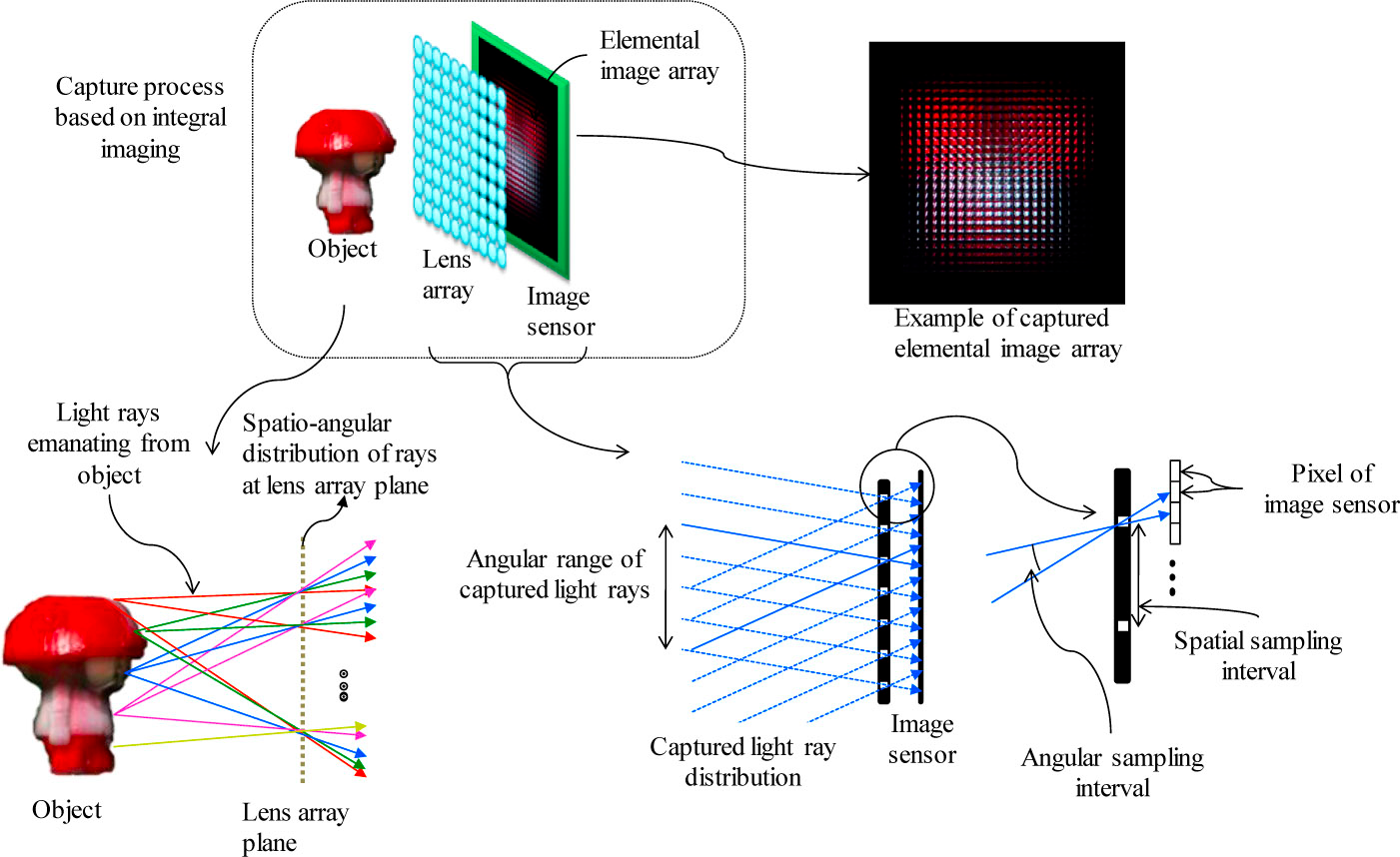
1. The program will ask you to select an Image. Provide the file path to the Image File. 
2. The program will then execute six commands in a linear fashion where at each command the user will provide input for that specific command. E.g.,

****

1. The last command will display the user supplied picture with the resulting adjustments.

**Related Work**

Image representation and compression is a well established field but the steady increase of advanced image types, such a integral imaging and light field technology, presents a challenge to todays modern compression and image representation algorithms. Image compression is becoming more mandatory for the storage and transmission of these advanced image types and algorithms are needed to address these concerns, such as the algorithm Aaron Aggoun’s explains in a November, 2011 issue of Journal of Display Technology where he describes “the use of the lifting scheme in the application of a 3D Wavelet Transform for the compression of 3D Integral Images”. The picture below is an example of three-dimensional information capture and manipulation using integral imaging compression.



**Conclusions**

The results of this work show that the highest compression ratios are achieved using the YCbCr color space. This leads to the conclusion that several color models such as RGB make the selective discarding of information very difficult because they tend to spread their useful visual image information evenly across each of their three color components. Also, in order to provide efficient and flexible compression some form of predictor selection is necessary.

**Bibliography**  
  
Drake, Fred L., Jr. "Python V2.7.3 Documentation." *Overview â”*. Python Software Foundation, 19 Sept. 2012. Web. 20 Sept. 2012. <[http://docs.python.org/>.](about:blank)  
  
"Python Imaging Library Handbook." *Python Imaging Library Handbook*. Pythonware.com, 6 May 2005. Web. 20 Sept. 2012. <http://www.pythonware.com/library/pil/handbook/index.htm>.  
  
Li, Ze-Nian, and Mark S. Drew. *Fundamentals of Multimedia*. Upper Saddle River, NJ: Pearson Prentice Hall, 2004. Print.  
   
A. Aggoun, "Compression of 3D Integral Images Using 3D Wavelet Transform," J. Display Technol. **7**, 586-592 (2011)  
<<http://www.opticsinfobase.org/jdt/abstract.cfm?URI=jdt-7-11-586>>

Heckbert, P. "Color Image Quantization for Frame Buffer Display", Computer

Graphics, Vol 16, #3, pp. 297-303, 1982

"Project: What's In A GIF - LZW Image Data." *3MF*. N.p., n.d. Web. 01 Nov. 2012. <http://www.matthewflickinger.com/lab/whatsinagif/lzw\_image\_data.asp>

**Appendix**:

***A1***: *Color reduction functions*

**def reduce**(color\_channels, width, S=((None, None), (None, None), (None, None))): # Use default values of None until implemented

"""

(Task 1): Given three channels c1, c2, c3 of an image, and three numbers s1, s2, s3 reduce the resolution of the channels of the image by 1/s1, 1/s2, 1/s3, respectively.

"""

c1, c2, c3 = color\_channels

if not all(s for SI in S for s in SI):

raise ValueError("Cannot reduce resolution by a None value.")

# return c1, c2, c3

if any(s <= 0 for SI in S for s in SI):

raise ValueError("Cannot reduce resolution by a negative or 0 value.")

s1, s2, s3 = S

# reduced channels

c1\_r = reduce\_channel(c1, width, \*s1)

c2\_r = reduce\_channel(c2, width, \*s2)

c3\_r = reduce\_channel(c3, width, \*s3)

return c1\_r, c2\_r, c3\_r

**def reduce\_channel**(channel, width, sx, sy):

height = len(channel) // width

new\_width = int(round(width / sx))

new\_height = int(round(height / sy))

# There are now scale x scale pixels per new pixel

new\_chan = []

new\_row = []

for row in range(new\_height):

for col in range(new\_width):

left = sx \* col

top = sy \* row

square = get\_square(channel, width, left, sx, top, sy)

if len(new\_row) >= new\_width:

new\_chan.append(tuple(new\_row))

new\_row = []

new\_row.append(sum(square) / len(square))

new\_chan.append(tuple(new\_row))

return new\_chan

**def get\_square**(channel, width, left, sq\_width, top, sq\_height):

rows = []

for i in range(sq\_height):

rows.extend(channel[((top + i) \* width) + left : ((top + i) \* width) + (left + sq\_width)])

return rows

***A2***: *Quantization algorithms*

**def quantize**(color\_channels, N=((None,)\*3)): # Default values of None to indicate no quantization.

"""

(Task 2): Given three channels c1, c2, c3 of an image, and three numbers n1, n2, n3,

uniformly quantize the channels into n1, n2, n3 bins respectively.

"""

c1, c2, c3 = color\_channels

n1, n2, n3 = N

c1New = calcquant(c1, n1)

c2New = calcquant(c2, n2)

c3New = calcquant(c3, n3)

return c1New, c2New, c3New

**def calcquant**(channel, numBins):

values = range(0, 256)

if numBins is None:

return channel

binsize = math.floor(len(values)/numBins)

offset = math.floor(binsize/2)

bins = dict()

lowerbound = 0

key = 0

median = lowerbound + offset

upperbound = lowerbound + binsize

while(upperbound < len(range(0, 256))):

while(lowerbound < upperbound):

bins[key] = median

key = key + 1

lowerbound = lowerbound + 1

upperbound = upperbound + binsize

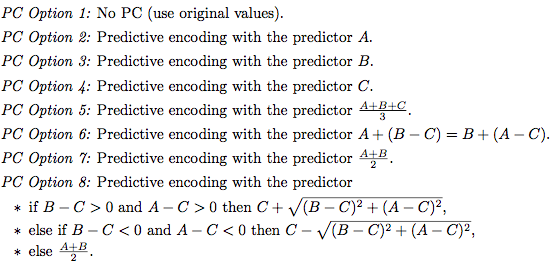
median = lowerbound + offset

quantized = list()

for i, val in enumerate(channel):

quantized.append(bins[val])

***A3*:** *Predictive coding schemes*

**

**def predict\_encoding**(r\_g\_b, widths, heights, choice):

r, g, b = r\_g\_b

r\_width, g\_width, b\_width = widths

r\_height, g\_height, b\_height = heights

r\_errors, r\_predicted = predict(r, r\_width, r\_height, choice)

g\_errors, g\_predicted = predict(g, g\_width, g\_height, choice)

b\_errors, b\_predicted = predict(b, b\_width, b\_height, choice)

#z\_new = zip (r\_errors,g\_errors,b\_errors)

z\_new = (r\_errors, g\_errors, b\_errors)

return z\_new

**def predict** (r, width, height, choice):

table = [[0 for i in range(width)]for j in range (height)]

table\_o = [[0 for i in range(width)]for j in range (height)]

table\_e = [[0 for i in range(width)]for j in range (height)]

# Creates 2 tables with (height) rows and (width) columns with all values initialized to 0

# table\_o (original) will hold original values from the list given

# table\_e (errors) will hold the error values.

for d1 in range (height):

table[d1][0] = r[d1\*width]

for d2 in range (width):

table [0][d2] = r[d2] # Now table is fully initialized - It has all of the top and left values from 'r'.

for d1 in range (height):

table\_e[d1][0] = r[d1\*width]

for d2 in range (width):

table\_e [0][d2] = r[d2]

# the first value of every row and column will match the originals, the other values will be

# error (predicted - original)

for d1 in range (height):

for d2 in range (width):

table\_o[d1][d2] = r[width\*d1 + d2]# now table\_o has all values from 'r'.

if (choice == 1):

return r,r

elif (choice == 2):

for row in range(1, height):

for column in range(1, width):

#table[row][column] = table[row][column-1]# if supposed to use prev. predicted values as predictors

table[row][column] = table\_o[row][column-1] # if supposed to use original values as predictors

table\_e[row][column]= table[row][column]-table\_o[row][column]

elif (choice == 3):

for row in range (1, height):

for column in range (1, width):

#table [row][column] = table[row-1][column]

table [row][column] = table\_o[row-1][column]

table\_e[row][column]= table[row][column]-table\_o[row][column]

elif (choice == 4):

for row in range (1, height):

for column in range (1, width):

#table [row][column] = table[row-1, column-1]

table [row][column] = table\_o[row-1][column-1]

table\_e[row][column]= table[row][column]-table\_o[row][column]

elif (choice == 5):

for row in range (1, height):

for column in range (1, width):

#table [row][column] = (table[row][column-1]+table[row-1][column]+table[row-1][column-1])/3

table [row][column] = (table\_o[row][column-1]+table\_o[row-1][column]+table\_o[row-1][column-1])/3

table\_e[row][column]= table[row][column]-table\_o[row][column]

elif (choice == 6):

for row in range (1, height):

for column in range (1, width):

#table [row][column] = (table[row][column-1])+(table[row-1][column]-table[row-1][column-1])

table [row][column] = (table\_o[row][column-1])+(table\_o[row-1][column]-table\_o[row-1][column-1])

table\_e[row][column]= table[row][column]-table\_o[row][column]

elif (choice == 7):

for row in range (1, height):

for column in range (1, width):

#table[row][column] = (table[row][column-1]+table[row-1][column])/2

table[row][column] = (table\_o[row][column-1]+table\_o[row-1][column])/2

table\_e[row][column]= table[row][column]-table\_o[row][column]

elif (choice == 8):

for row in range (1, height):

for column in range (1, width):

#a,b,c = table[row][column-1],table[row-1][column],table[row-1][column-1]

a,b,c = table\_o[row][column-1],table\_o[row-1][column],table\_o[row-1][column-1]

if ((b-c)>0 and (a-c)>0):

table[row][column]= c + int(sqrt((b-c)\*(b-c)+(a-c)\*(a-c)))

table\_e[row][column]= table[row][column]-table\_o[row][column]

elif ((b-c)<0 and (a-c)<0):

table[row][column] = c - int(sqrt((b-c)\*(b-c)+(a-c)\*(a-c)))

table\_e[row][column]= table[row][column]-table\_o[row][column]

else:

table[row][column] = (a+b)/2

table\_e[row][column]= table[row][column]-table\_o[row][column]

else:

print "error"

return\_list\_predicted = []

return\_list\_errors = []

for row in range (0, height): #Go through table in order, and add all values to return\_list

for column in range (0, width):

n = table[row][column]

return\_list\_predicted.append(n)

for row in range (0, height): #Go through table in order, and add all values to return\_list

for column in range (0, width):

n = table\_e[row][column]

return\_list\_errors.append(n)

return (return\_list\_errors, return\_list\_predicted)

***A4***: *Shannon-Fano Encoding*

**def encodeShannonFanno**(chan, codTbl):

codedChan = ""

for val in chan:

for entry in codTbl:

if(val == entry.val):

codedChan += entry.getCode()

break

return codedChan

#evaluate list of colors

**def evaluateFrequencies**( chan):

v\_list = [] #for tracking colors seen in 'chan'

sf\_entArr = [] #list of Color Entries. Recall that ColorEntry tracks frequency

for val in chan: #for each color listing in chan

if( val not in v\_list):

v\_list.append( val) #track this val

sf\_ent = Sf\_Entry(val) #start new val entry

sf\_entArr.append( sf\_ent) #add val entry to list

##print( 'Found new val:{0}'.format( val))

sf\_entArr[ v\_list.index( val)].incrFreq() #increase frequency

return sf\_entArr

#sort list by frequency in descending order

**def sort**( arr, ascending):

#quicksort

if( len( arr) <= 1):

return arr

pivot = arr.pop( len( arr)//2) #get middle entry as pivot

upper = []

lower = []

arrOut = []

#print(arr)

for sf\_ent in arr:

if(sf\_ent.freq > pivot.freq):

upper.append( sf\_ent)

else:

lower.append( sf\_ent)

if(ascending):

arrOut.extend( sort( lower, ascending))

arrOut.append( pivot)

arrOut.extend( sort( upper, ascending))

else:

arrOut.extend( sort( upper, ascending))

arrOut.append( pivot)

arrOut.extend( sort( lower, ascending))

#print("\nIter on length {0}, done. Returning this:".format(len(arr)))

#print (arrOut)

return arrOut

#use list slicing here, and value as we slice

**def segmentTable**( tbl):

tbl\_buffer = tbl

code\_buffer = 0

top\_tbl = []

bot\_tbl = []

allsum = 0

topsum = 0;

botsum = 0

for shafa\_code in tbl\_buffer:

allsum = allsum + shafa\_code.freq

##print('========================\n\nSum of all frequencies = {0}\n'.format(allsum))

#break table into two segments with roughly equal frequency sums

while(topsum < allsum // 2) and (len( tbl\_buffer) > 0):

buffer0 = tbl\_buffer[0]

topsum += buffer0.freq

top\_tbl.append( tbl\_buffer.pop( 0)) #push Code Entry to top list

## print(top\_tbl)

#do the same with bottom segment, if bottom segment 'sum < halfSumAllFrequencies'. these tables will be roughly equal

while(botsum < allsum // 2) and (len( tbl\_buffer) > 0):

buffer1 = tbl\_buffer[0] #pop another element off the list. We're nwo in lower table's half pretty much.

botsum += buffer1.freq

bot\_tbl.append( tbl\_buffer.pop( 0)) #push Code Entry to top list

if(botsum > allsum):

lastInd = len( top\_tbl) - 1

lastTopEntry = top\_tbl[ lastInd]

if(botsum + lastTopEntry.freq < topsum) and (botsum + lastTopEntry.freq > (allsum / 2)):

bot\_tbl.append( top\_tbl.pop( lastind))

#check if last entry from top segment plus bottom sum is greater than half the total sum.

for entry in top\_tbl:

## print("\n{0} Appended 0".format( entry.printColor()))

entry.appendToCode(0) #left shift top codes and set LSB to 0

for entry in bot\_tbl:

## print("\n{0} Appended 1".format( entry.printColor()))

entry.appendToCode(1) #left shift top codes and set LSB to 1

if len(top\_tbl) > 1:

top\_tbl = segmentTable( top\_tbl)

if len(bot\_tbl) > 1:

bot\_tbl = segmentTable( bot\_tbl)

top\_tbl.extend(bot\_tbl)

return top\_tbl

**A5**: LZW Encoding

**def lzw\_encode**(seq):

seq = list(seq) # make a copy

seq.reverse() # For efficiency, we will reverse the list and get data from the end

output = []

dictionary = {str(s) : s for s in range(256)}

i = 256

try:

s = seq.pop()

except IndexError:

return output

# dictionary = [] # TODO: Set a default set of values for the dictionary

while True:

try:

c = seq.pop()

except IndexError:

break

if (s + c) in dictionary:

s = s + c

else:

output.append(dictionary[s])

dictionary[s + c] = i

i += 1

s = c

# if len(output) == 46643:

# import pdb; pdb.set\_trace()

if s in dictionary:

output.append(dictionary[s])

else:

output.append(i)

return output

**def lzw\_decode**(seq):

seq = list(seq)

seq.reverse()

output = []

dictionary = {s : str(s) for s in range(256)}

i = 256

try:

s = (dictionary[seq.pop()], )

except IndexError:

return output

output.append(s[0])

while True:

try:

k = seq.pop()

except IndexError:

break

if k in dictionary:

entry = dictionary[k]

elif k == i:

entry = s + (s[0], )

if isinstance(entry, str):

output.append(entry)

dictionary[i] = s + (entry, )

i += 1

else:

output.extend(entry)

dictionary[i] = s + (entry[0], )

i += 1

if isinstance(entry, str):

s = (entry, )

else:

s = entry

return output

Contributions  
Steven Carr - Documentation, code review and testing  
Casey Kuball – Task I & Task VI  
Glenn Craver – Task V  
Wesley Rose - Task III

Dawne Flanagan - Task II & Task IV