Project 3 – Compression

Overview: In this project, you will be creating code to compress and decompress image files.

Goal: To gain experience with compression algorithms.

Requirements: You are going to implement a compression algorithm in Python 3.x. You have been given a set of example images and your goal is to compress them as much as possible without losing any perceptible information – upon decompression they should appear identical to the original images.

Images are essentially stored as a series of points of color, where each point is represented as a combination of red, green and blue (rgb). Each component of the rgb value ranges between 0-255, so for example: (100, 0, 200) would represent a shade of purple. Using a fixed length encoding, each component of the rgb value requires 8 bits to encode ($2^8 = 256$) meaning that the entire rgb value requires 24 bits to encode. You could use a compression algorithm like Huffman encoding to reduce the number of bits needed for more common values and thereby reduce the total number of bits needed to encode your image.

You can use the Pillow library of Python to read in image files and extra the rgb values of individual points. To install the library in PyCharm, go to Project Settings -> Project Interpreter. To the right of the list of packages will be a small + sign. Click on it to open up the window of available packages and search for Pillow – since there are several similarly named libraries, you want the one written by Alex Clark and its full name should be Python Imaging Library (Fork). Then click on Install Package. (It's possible you may get an error if you have an older version of pip installed – it would be listed on the prior page of installed packages. If the error occurs, first install the latest version of pip, then install Pillow.)

Once you have Pillow installed, you can include the library into your project code with:
```
from PIL import Image
```
You can then open image files using, for example:
```
img = Image.open("powercat.bmp")
```
If you wanted to display the image in your standard image viewer, you could use:
```
img.show()
```
If you want to access the pixel values of points in an image, you'll first need to call load() to create a pixel access object and then you can access individual pixels using list notation:
```
width, height = img.size
px = img.load()
for x in range(width):
    for y in range(height):
        print(px[x,y])
```

Your code should read in an image file, compute how many bits a required for a fixed length encoding and then apply a compression algorithm to create a smaller encoding – you need to implement the compression, you cannot use a compression library. You should output how many bits are required to store the image in your compressed format as well as the compression ratio achieved. When it comes to saving your compressed image, you won't be able to save it as a standard image format, since you will have created your own encoding, but you can save the sequence of bits into a text file.

Your code should also be able to prompt the user for the filename of a text file containing a compressed sequence of bits and then decompress that file into the original image – you can assume that the file uses the same compression format as the last file you compressed. So, for example, if you compressed pacificat.bmp into a series of bits stored in pacificat.txt and then the user asked you to decompress alt_pacificat.txt, you could assume that alt_pacificat.txt used the same compression data structure as pacificat.txt (it might be a subset of the data from the original image, for example).

Extra Credit Opportunity: there are a number of libraries that can help you store formatted data into a file from Python. If you research the options and find a way to store your compression data structure into a file, such that the user can select both a bit file and a data structure file and use the data structure to decompress the bit file, then you can earn a small amount of extra credit.

Your code should compute the following statistics each time the ***compression*** code is called on a file:

- Runtime in milliseconds of the compression process, including any time needed to create the data structures used to help with the compression process.
- Number of bits needed by your algorithm to encode the contents of the file.
- Number of bits needed by a fixed length encoding to encode the contents of the file.
- The compression ratio achieved.

For the ***decompression*** process, you only need to track the runtime in milliseconds.

Algorithm: In addition to your code, you need to turn in a well-formatted document containing the pseudo-code for your core algorithm (written in the same format as the textbook uses for pseudo-code) along with any proofs you decide to include about your algorithm – see grading section below.

Grading: Your algorithm will be graded according to the rubric below.

	1	**2**	**3**	**4**
Compression	Can compress files to be consistently smaller than original.	Compressed size at least as good as Huffman.	Compressed size at least as good as benchmark code.	Best compression ratio of all student submissions.
Decompression	Inaccurate decompression.	Decompressed image is close but perceptibly different.	Recreates image either exactly as before, or so close as to be	

			imperceptible to the human eye in good light*.	
Time	Code always terminates in finite time.	Runtime of compression and decompression code no worse than clean implementation of Huffman.	Best compression and decompression time performance of all student submissions, while still able to accurately decompress.	
Efficiency Analysis	Defines input size and basic operation for domain. States without proving what efficiency class the algorithm belongs to.	Defines input size and basic operation for domain. Proves that algorithm belongs to Θ(g(n)).		

* For the purposes of grading, we will consider "imperceptibly different" to mean that all pixel values are within 2.5% of the original. If you can find documentation of a different and measurable standard, then you are free to use that standard and submit your documentation along with your code submission.

Your project will receive a final score based on the number of points from the above rubric that it accrues. The table below summarizes the minimum number of points from the rubric that would be necessary to earn various grades. Extra credit is possible.

Grade:	50	45	40	35	30
Points on Rubric:	10	9	8	7	6

Runtime Guides:
The following benchmarks can be used in determining whether your code is matching the performance of a clean Huffman implementation and the benchmark algorithm. The benchmark times were based on running the code on one of the lab computers. The times are averages over x runs.

Due to the campus closure, it hasn't been possible to test on the lab computers. For now, I've included compression runtimes from my computer, which has tended to have runtimes reasonably comparable to the lab computers on past assignments.

File	**Huffman**		**Benchmark**	
	Compressed Size (in number of bits) & (percent of fixed)	**Compression & Decompression Runtimes (in msecs)**	**Compressed Size (in number of bits) & (percent of fixed)**	**Compression & Decompression Runtimes (in msecs)**
small.bmp	76 19.79%	15.625	76 19.79%	15.625
flag.bmp	14,293,189 32.81%	1500.035	14,266,734 32.75%	2734.430
powercat.bmp	5,398,861 85.67%	249.961	4,765,151 75.61%	531.245
colors.bmp	10,957,824 97.56%	468.767	9,760,083 86.90%	843.710
burns.bmp	3,940,233 98.02%	156.249	3,451,786 85.87%	328.131
fall.bmp	38,827,575 87.77%	1515.644	35,865,447 81.08%	2859.387
knoles.bmp	7,216,763 92.52%	296.865	6,794,839 87.11%	531.260
ocean.bmp	46,551,267 95.59%	1734.389	40,856,664 83.89%	3499.941