CPSC 217 Assignment #4: Image Compression

Due: Monday, June 16, 2025 at 11:30pm
Weight: 7%
Sample Solution Length: Approximately 100 lines (with only minimal comments)

Individual Work:

All assignments in this course are to be completed individually. Use of large language models, such as ChatGPT, and/or other generative AI systems is prohibited. Students are advised to read the guidelines for avoiding plagiarism located on the course website. Students are also advised that electronic tools may be used to detect plagiarism.

Late Penalty:

Late assignments will not be accepted.

Submission Instructions:

Submit your .py file electronically to the Assignment 4 drop box in D2L. You don't need to submit the image files or SimpleGraphics.py – we already have them.

Description

Widely used image storage formats like PNG and JPEG use a variety of techniques to store image data compactly. While the small file sizes that these formats provide are desirable, this benefit comes at the cost of significant complexity in both the encoders and decoders. This makes them difficult to implement correctly and, at least in some cases, computationally expensive to run.

Dominic Szablewski created the Quite OK Image format which he first described in a blog post on November 24, 2021[1]. The format that he created uses some relatively straightforward techniques to quickly compress images. While the resulting files are not quite as small as well-compressed PNG files, the lack of complexity and better performance outweighs the file size penalty in some contexts. The Quite OK Image format also has the benefit that it is not encumbered by any patents, and the code for loading and saving images is distributed under a permissive license that allows it to be used in both personal and commercial projects without payment. A second blog post on December 20, 2021, described the finalized format and required only a single page to do so[2].

The Quite OK Image format is a binary file format. While binary files can be read and written with Python programs in much the same way that text files can, there are some additional details that need to be considered that we won't discuss in this course. As such, I have created a text file format for storing image data that is a variant of the Quite OK Image format for use in this assignment. I will refer to it as the OK Text Image (OKTI) format. While text file image formats are rare, they do exist. For example, the portable pixel map (PPM), portable gray map (PGB), and portable bit map (PBM) formats have both binary and text file variants which store full color, gray scale, and pure black and white images respectively.

[1] https://phoboslab.org/log/2021/11/qoi-fast-lossless-image-compression
[2] https://phoboslab.org/log/2021/12/qoi-specification

In this assignment, you will write a function to encode images in the OKTI format. Then you will use that function to write a program that loads an image stored in common image format (PNG or PPM) and stores it as an OKTI image. A detailed description of the image format can be found in the next section. Additional requirements for your program are described in later sections.

OKTI File Format

The OKTI format begins with a line containing nothing but the string "okti". This allows one to quickly determine whether or not a file contains OKTI data. The second line in the file contains two positive integers separated by a space. These positive integers are the width and height of the image, respectively.

The remainder of the file describes all of the pixels in the image. Pixels in the OKTI format are encoded in order from the upper left corner of the image to the lower right corner of the image, moving across each row in the image (from left to right) with the rows encoded from top to bottom. Each pixel can be represented in one of 4 ways:

- As a full set of RGB values
- As the difference in RBG values from the immediately previous pixel
- As a run of several copies of the immediately previous pixel
- As a copy of a previously encountered pixel (not just the immediate predecessor)

Full RGB values

Pixels encoded as a full set of RGB values use 7 characters. The first character is always a lowercase ‘p’. This character is immediately followed by 6 hexadecimal digits. The first two encode the amount of red, the middle two encode the amount of green, and the final two encode the amount of blue. All three color components can range from 0 to (and including) 255 (which is denoted by ff). For example, a line in the file consisting of pff0000 represents one bright red pixel while p808080 represents a pixel that is middle gray colored.

Difference in RGB Values

When the current pixel is similar in color to the immediately previous pixel it is more efficient to encode it as a difference from the previous pixel rather than as a full set of RGB values. A pixel encoded in this manner begins with a lowercase ‘d’, followed by 3 hexadecimal digits. The first digit is the difference in the red component, the second digit is the difference in the green component, and the final digit is the difference in the blue component. Each of these values is stored with a bias of 8 meaning that the digit is 8 larger than the actual difference. For example, d08f is used to indicate that the current pixel’s red value is 8 smaller than the previous pixel’s (because 0 - 8 = -8), its green value is the same as the previous pixel’s (because 8 - 8 = 0), and the blue value is 7 larger than the previous pixel’s (because f - 8 = 7). Similarly, da27 is used to indicate that the current pixel’s red value is 2 larger than the previous pixel’s (because a - 8 = 2), the current pixel’s green value is 6 smaller than the previous pixel’s (because 2 - 8 = -6), and the blue value is 1 smaller than the previous pixel’s (because 7 - 8 = -1).

The previous pixel should be initialized to black (r: 0, g: 0, b: 0) before loading begins. This initial value allows the first pixel in the image to be a difference from a previous pixel.

A Run of Several Pixels

There are two variations of pixel runs, both of which encode copies of the immediately preceding pixel. The first variation consists of a lowercase ‘r’, followed by a single hexadecimal digit that represents the number of additional copies of the previous pixel that should be included in the image. This allows up to 15 copies of the previous pixel to be represented. For example, r5 is used to indicate that 5 additional copies of the previous pixel should be included in the image while re indicates 14 additional copies. The second variation consists of an uppercase ‘R’, followed by two hexadecimal digits that represent the number of additional copies of the previous pixel. This allows up to 255 copies of the previous pixel to be represented. For example, R20 represents 32 copies of the previous pixel while Rfe represents 254 copies of such. Note that it is possible that a run will wrap around onto the next row of the image, and in extreme cases, a run could span several rows.

The previous pixel should be initialized to black (r: 0, g: 0, b: 0) before loading begins. This initial value allows the first pixel in the image to be a run of previous pixels.

Copy of a Previous Pixel

Each time a pixel is added to the image its color should be inserted at the front of a list of previously encountered colors **only if it is not already present in that list**. (As a consequence, the list will never contain any duplicate values). This list will store a maximum of 256 previous colors. If adding an additional color at the front of the list will violate this restriction, then the last color in the list should be discarded as the new value is inserted so that its length remains 256.

Like runs of pixels, there are two variations for copying a previous pixel. The first variation consists of a lowercase ‘i', followed by a single hexadecimal digit which is the index into the list of previous colors. For example, i0 is used to indicate that the current pixel should have the color that is stored at index 0 in the list, while ib indicates the current pixel’s color can be found at index 11 in the list. The second variation consists of an uppercase ‘I’, followed by two hexadecimal digits that are the index into the list of previous colors. For example, I11 is used to indicate that the current pixel will have the color that is stored at index 17 in the list, while Ia2 indicates that the current pixel’s color is at index 162 in the list.

The list of previous colors should initially contain one entry representing black (r: 0, g: 0, b: 0). This initial value allows the first pixel in the image to be a copy of a previous pixel.

Encoder Requirements

You must create a function that saves all of the pixels in an image object into a file in the OK Text Image format (as described previously). Your function will take two arguments which will be the image object to save and the name to use when saving it. When saving the image, your function should encode the file as compactly as possible. This can be accomplished by prioritizing the encoding of pixels in this order:

- As a run of one or more copies of the immediately previous pixel
- As a pixel of a previously encountered color
- As a difference from the immediately previous pixel’s color
- As a full set of RGB values

Which encoding is used will be dictated by the color of the current pixel. For example, a pixel can only be encoded as a run of one or more copies of the previous pixel if its color matches that of the

immediately previous pixel; a pixel can only be encoded as a pixel of a previously encountered color if its color matches one of the 256 most recently encountered colors; a pixel can only be encoded as a difference from the immediately previous pixel's color if the difference in RGB values is sufficiently small. If none of those constraints are met, then the pixel must be encoded as a full set of RGB values.

You may be tempted to use nested for loops to visit every pixel in the image and encode them in sequence. While such an approach works well for pixels that repeat a previous color, are encoded as a difference from the immediately preceding pixel, or are encoded as a complete set of RGB values, this approach does **not** work well for runs of several identical pixels. As a result, I'd suggest using a while loop instead of a for loop. Initialize variables that represent the x and y location in the image ahead of the loop. The body of the loop will increase the x value as each pixel is processed. If the x value reaches the width of the image, reset the x value to 0 and increase the y value by 1. The loop should continue to execute until the y value has exceeded the bottom row of the image. In many cases the value of x will only increase by 1 each time the loop's body executes, but it will increase by a larger value when a run of pixels is encountered (and the run may also result in the x value being reset to 0 and the y value being increased).

Main Program

Once you have developed the encoder described previously, you must create a main program that demonstrates its functionality. If the user provides one command line argument when the program is run, then that command line argument will be the name of the existing PPM or PNG image file that will be opened. If no command line arguments are provided, then your program should read the name of the file from the user using the input function. Otherwise, your program should report that too many command line arguments were provided and quit.

Once the file name for the image has been retrieved from the command line or read from the user, the file should be opened and displayed by using the loadImage and drawImage functions. Then your encoding function should be called to write out the equivalent image data in the OKTI format. Save the OKTI image data in a file named output.okti.

When the loadImage function is called on a file that it can decode successfully it returns an image object (which can subsequently be drawn by calling drawImage). If the file cannot be loaded successfully (either because it doesn't exist or because it doesn't contain data that is in a supported format) then a _tkinter.TclError exception is raised. If this exception is raised, your program should catch the exception, display an appropriate error message, and quit. You will need to import _tkinter near the top of your program to catch this exception because this type of exception is defined in that module.

The images have been provided in both PPM and PNG formats because some Python installations do not include the necessary libraries to decode PNG files, but PNG files are more widely used and are viewable in a web browser. It's fine if your program only works for PPM files because you don't have the necessary libraries for decoding PNG files installed.

Working with Hexadecimal Values

The OKTI format makes heavy use of hexadecimal values. While you could create your own function for converting from an integer to a hexadecimal value, doing so is unnecessary because Python already includes the necessary functionality. Hexadecimal character sequences can be created using Python's formatting operator, which is denoted by %. The %x format is used to indicate that an integer should be

formatted in base 16. For example, if i is a variable that contains 30 then "%x" % i will evaluate to the string "1e" (the hexadecimal representation of 30). In some instances, you may want to ensure that the hexadecimal digit sequence generated using the format operator includes two digits, including a leading 0 if necessary. This can be accomplished using the string "%02x" to the left of the format operator. For example, if i is 11 then "%02x" % i will evaluate to "0b".

Other Reminders

Strings can be concatenated (joined together) with the + operator. You may find this helpful when constructing the character sequences needed for some (or all) of the pixel types.

Hints:

Develop your encoding function incrementally. Start by implementing support for only full RGB values. Once that works, add support for pixels that are represented by a difference from the previous pixel. Then add support for runs of identical pixels. Finally, add support for the list of previous pixels. Test images have been provided that only use full RGB values and full RGB values plus one other pixel type to help you focus on one type of pixel at a time.

You will be writing files as part of this assignment. Always double check that the file name you have provided is correct before writing as it is easy to inadvertently overwrite an existing file, and once this occurs, there is no way to recover the original file. **You are strongly advised to keep a backup of your Python file** in a different folder, or better yet, on a different device in case you inadvertently overwrite it. One easy way to accomplish this is to occasionally email a copy of your program to yourself. You will always write your output to a file named output.okti. Use this string directly in your program instead of reading the file name from the user to minimize the risk of overwriting a different file than intended.

The diff utility (which is available on the Linux machines in the tutorial rooms) can be used to quickly determine what differences, if any, are present between two files. On Windows, the file comparison utility is named fc. It can be run from the windows command prompt. If you save the expected output file from the course website in the same folder as your program, you can quickly determine whether or not your program's output matches the expected output using these utilities. Alternatively, there are many websites that will identify the differences between two files. Copying and pasting (or uploading) your program's output and the expected output to such sites is also an option for verifying that your program's output matches what is expected.

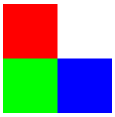
Sample Images:

Numerous sample images have been provided in PNG, PPM and OKTI formats. Your program should load either the PNG or PPM file. The output your program produces should match the provided OKTI file exactly (other than possible differences in how the end of a line is represented). The images described in this section were specifically constructed so that they use only a limited selection of pixel types which will allow you to focus on each pixel type individually.

Small Images:

These images contain a relatively small number of pixels, which makes them faster work with and limits the amount of output that is generated if your program includes print statements to assist with debugging. While not strictly required, I'd recommend starting with these images.

tiny.ppm / tiny.png / tiny.okti

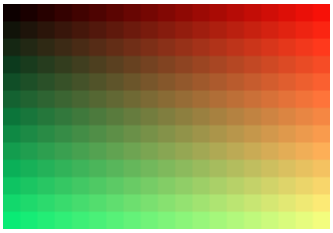


This image consists of only 4 pixels. The lines required to represent it in the OKTI format are shown below.

```
okti
2 2
pff0000
pffffff
p00ff00
p0000ff
```

The first line in the file marks that the file contains OKTI data. The second line in the file is "2 2" because the file contains 2 columns and 2 rows. The bright red pixel is represented by pff0000, the white pixel is represented by pffffff, the green pixel is represented by p00ff00, and the blue pixel is represented by p0000ff. The pixels appear in the order shown because the image is encoded from left to right within each row, and the rows are encoded from top to bottom.

pixels_small.ppm / pixels_small.png / pixels_small.okti

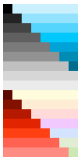


This image has 13 rows and 19 columns. As a result, its first two lines are "okti" and "19 13". (The 19 comes first because it's the width of the image, the 13 comes second because it's the height of the image.) This is followed by 247 lines (because 13 * 19 = 247) that begin with "p", each of which encodes one pixel.

differences_small.ppm / differences_small.png / differences_small.okti

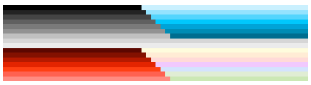
This image is 12 pixels wide and 10 pixels tall. As a result, its first two lines are "okti" and "12 10". It only makes use of p and d pixels.

runs_short_small.ppm / runs_short_small.png / runs_short_small.okti



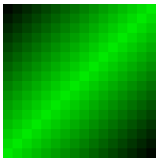
This image is 8 pixels wide and 16 pixels tall. It only makes use of p and r pixels.

runs_long_small.ppm / runs_long_small.png / runs_long_small.okti



This image is 64 pixels wide and 16 pixels tall. It only makes use of p and R pixels.

smallindices_small.ppm / smallindices_small.png / smallindices_small.okti



This image is 16 pixels wide and 16 pixels tall. It only makes use of p and i pixels.

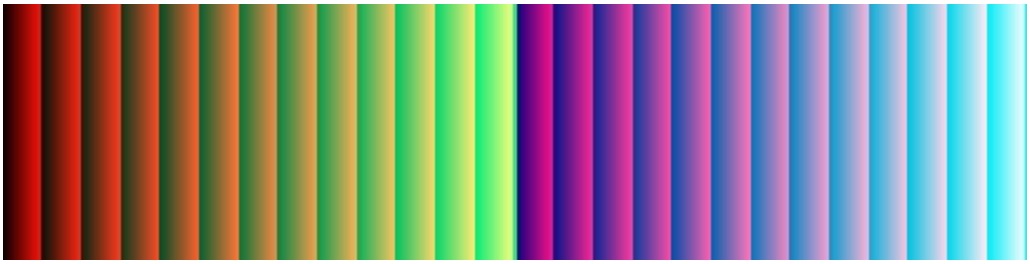
runs_and_short_indices_small.ppm / runs_and_short_indices_small.png / runs_and_short_indices_small.okti

This image is 8 pixels wide and 16 pixels tall. It makes use of p, i, and r pixels.

Larger Images:

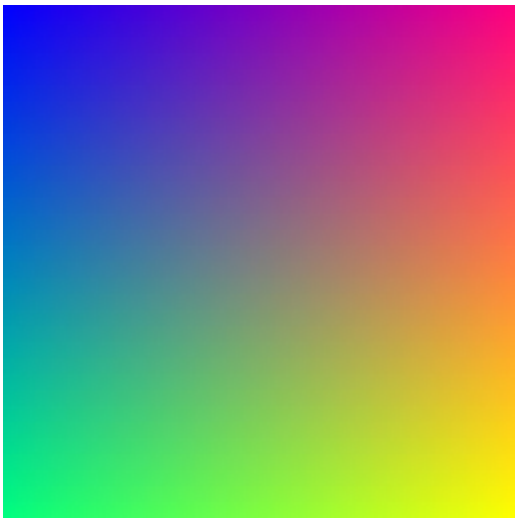
The images in this section are larger and could reveal bugs in your program that were not revealed by the smaller images. Like the images in the previous section, these images were carefully constructed so that they only use a limited selection of pixel types.

pixels.ppm / pixels.png / pixels.okti



This image is constructed only using pixels where a full set of RGB values are provided. It does not include any pixels represented as differences, runs, or indices into the list of previously encountered colors.

differences.ppm / differences.png / differences.okti



Two types of pixels are used in this image: full sets of RGB values and differences from the immediately previous pixel. It does not include any runs or indices into the list of previously encountered colors.

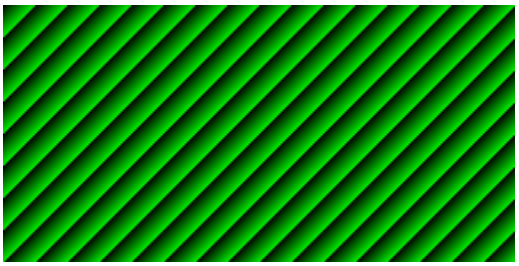
shortruns.ppm / shortruns.png / shortruns.okti

Only pixels represented as a full set of RGB values and runs of the immediately previous pixel are used in this image. All of the runs are less than 16 pixels in length which allows their length to be represented using a single hexadecimal digit.

longruns.ppm / longruns.png / longruns.okti

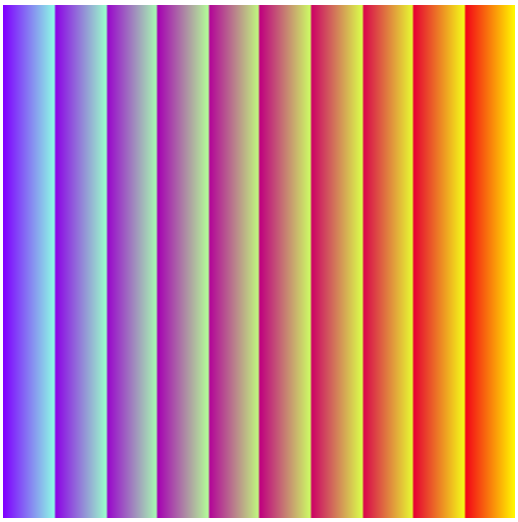
Like the previous image, this one only uses pixels where a full set of RGB values are provided and runs of the immediately previous pixel, but instead of using runs of 1 to 15 pixels in uses only runs that are at least 16 pixels in length (and as such, the length of the run is specified using two hexadecimal digits instead of one).

smallindices.ppm / smallindices.png / smallindices.okti



All of the pixels in this image are represented as either a full set of RGB values or a copy of a pixel that has been encountered previously. When a pixel is represented as a copy of a previous pixel, it is always a pixel that is among the 16 most recently encountered pixels which allows its position in the list of previous pixels to be represented by a single hexadecimal digit.

indices.ppm / indices.png / indices.okti



Most of the pixels in this image are represented as indices into the list of previously encountered in pixels. A few of these pixels use a single digit for the index into the list of pixels while most use a two-digit index. This image does not include any pixels represented as a run or a difference from the immediately previous pixel.

Additional images are available on the course website which use all of the pixel types. Ensure that your program also works correctly with those images.

Additional Requirements:

- Include your name, student ID number, and a brief description of your program at the top of your .py file.
- If one command line argument is provided, then your program should use that command line argument as the name of the file to open. If no command line argument is provided, then your program should read the name of the file from the user by calling the input function. Otherwise, your program should report that too many command line arguments were provided and quit.
- Your program must verify that the file provided by the user exists and can be opened successfully by the loadImage function. If the file does not exist or cannot be opened by loadImage, then your program should display an appropriate error message and quit.
- All lines of code that you write must be inside functions (except for the function definitions themselves and any constant definitions). Make appropriate use of a main function.
- Close every file that you open.
- Your encoder must reside in a function that is separate from the main program that processes the command line argument or reads the file name from the user and opens the image file.

- Do **not** define one function inside of another function.
- Include appropriate comments on each of your functions. All of your functions (except for main) should begin with a comment that briefly describes the purpose of the function, along with every parameter and every return value.
- Your program must not use global variables (except for constant values that are never changed).
- Your program must use good programming style. This includes things like appropriate variable names, good comments, minimizing the use of magic numbers, etc.
- Break and continue are generally considered bad form. As a result, you are NOT allowed to use them when creating your solution to this assignment. In general, their use can be avoided by using a combination of if statements and writing better conditions on your while loops.

Grading

This assignment will be graded on a combination of functionality and style. A base grade will be determined from the general level of functionality of the program (Does the encoder handle each of the required pixel types? Does the main program read the file name from the user successfully? Is the necessary error checking performed? Etc.). The base grade will be recorded as a mark out of 12.

Style will be graded on a subtractive scale from 0 to -3. For example, an assignment which receives a base grade of 12 (A), but has several stylistic problems (such as magic numbers, missing comments, etc.) resulting in a -2 adjustment will receive an overall grade of 10 (B+). Fractional marks will be rounded to the closest integer.

Total Score (Out of 12)	Letter Grade
12	A
11	A-
10	B+
9	B
8	B-
7	C+
6	C
5	C-
4	D+
3	D
0-2	F

For an A+:

Like the Quite OK Image format developed by Dominic Szablewski, my OK Text Image format is designed to provide some compression with relatively little complexity rather than aggressively minimizing the size of the compressed data. One aspect of that minimal complexity for the OKTI format includes having each pixel (or group of pixels in the case of a run) stored on its own line. While this makes the files particularly easy to read and write, it imposes a space overhead of one end-of-line marker for each pixel. When these end-of-line markers are a single character they represent at least 12.5% of the image data stored in the file. In the worst case (two character end-of-line markers and a file that is stored almost entirely as runs) they represent approximately 40% of the data stored in the file. However, it isn't

actually necessary to store each pixel (or group of pixels in the case of a run) on its own line because all of the techniques used to represent the pixels are of fixed length, and as such, the end-of-line marker isn't needed to identify the end of one pixel and the beginning of the next.

For an A+, update your saving function so that it only includes newline characters between pixels when adding another pixel to the current line would cause it to be more than 79 characters in length.

Including only the end-of-line characters necessary to keep the lines in the file to less than 80 characters reduces the space overhead for the newline characters to less than 3% of the image data in the worst case, while still producing files that can be opened in most text editors without any of the lines wrapping. Additional test files are available on the course website to allow you to test your improved function.