CSC 236 Data Structures

Objectives

- deepen comfort in use of classes
- use triply nested lists, considering memory management issues of shallow copy vs. deep copy
- design algorithms for encryption/decryption and compute Big-O for each

---

Note that this is an assignment with larger scope, so it is labelled as a lab. It is advisable to begin working on it early.

Lab L01: Embedding Messages in Images

As we know from our work in CSC 226, images can be represented as a two dimensional matrix of pixels where each pixel in a color image has three channels: red green and blue (RGB); it is the relative intensities of these colors that make up the visible image that we see.

Steganography is the art of concealing a message, image, or file within another message, image, or file. The word steganography combines the Ancient Greek words steganos (στεγανός), meaning "covered, concealed, or protected", and graphein (γράφειν) meaning "writing".
(**http://en.wikipedia.org/wiki/Steganography**)

It is possible to change the intensity of a certain color channel in such a way as to hide an image inside of another one. For example, by dramatically reducing the levels of blue and green from the image below and setting the red intensities to random values, it effectively "hides" what the true one is under red "noise":

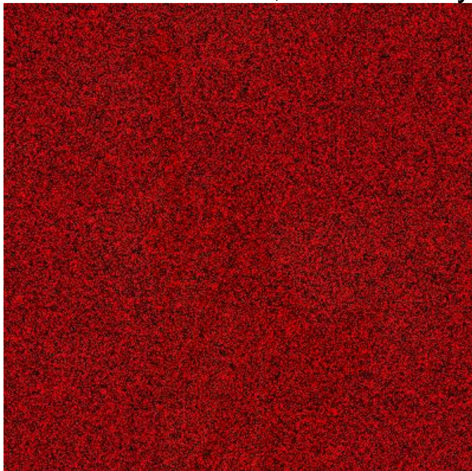


Image from a "nifty" assignments at Stanford University at http://nifty.stanford.edu/2011/parlante-image-puzzle/

If we boost the blue and green colors for all the pixels and set the red color values to zero, thereby ridding the image of the red noise, we get the following image:

Cool, isn't it?!!

Of course, it is possible to do the opposite in a sense and imbed a message inside of an image that makes it look like "noise". For example, take the image bc-sign that we used in CSC226 last year, which is shown above. The dimensions of this picture are 200 pixels by 125 pixels, or 25,000, which will be relevant later.

Embedding Technique 1

We can select one of the RGB values for each pixel, and embed a string inside that image as follows. Suppose you wanted to embed the text "Chapter 1", which is a string of characters "C", "h", "a", and so forth into the image. These 9

characters have an ordinal value representing their ASCII code, where "C" is 67, "h" is 104, "a" is 97 and so forth.

You can look at the "ASCII printable characters" table at **http://www.ascii-code.com/** for the character-ordinal value conversions or use the Python `ord()` command on any character as we did in CSC 226.

After conversion, the string "Chapter 1" becomes a sequence of integers, namely:
`67,104,97,112,116,101,114,32,49`
We can simply replace each pixel's blue values with these integers. As an example, we show below and to the left the original red, green, blue values for the first 9 pixels in the bc-sign image, with the modified version below:

```
Original RGB Values
251 253 251
252 249 233
243 232 187
235 215 142
233 208 124
248 222 128
239 218 107
238 221 98
232 212 90

RBG Values with BLUE replaced
251 253 67
252 249 104
243 232 97
235 215 112
233 208 116
248 222 101
239 218 114
238 221 32
232 212 49
```

Applying this trick and embed the first 20,000 characters of "Anna Karenina" to the "bc-sign" image, we get the one here.

Given that the ASCII values are within a narrow band (32 through 126) rather than the full range of possible values from 0 to 255, it kind of looks like the image

was given an odd "blue wash", where the blue values are muddled a little bit; some places look bluer than they should be and others look less so. To make matters worse, only part of this picture has this wash. If you did not know that this image has a message inside the blue channel, you may dismiss the oddity as corruption instead of a hidden message, which could be anything (such as the source code of a program or the schematics of a new carbon free engine). Note that the reason that only part of the image changed is that the picture itself consists of 25,000 pixels, and the blue saturation values for only the first 20,000 were changed.

Improved Embedding

After some thought, the manipulation of the image seems too obvious, so a less noticeable alternative is to take the average of the original Blue saturation values per pixel and the ordinal values of the characters in the message.

Applying this technique to the first nine pixels, we have the following information:

```
Original Pixel Values
251 253 251
252 249 233
243 232 187
235 215 142
233 208 124
248 222 128
239 218 107
238 221 98
232 212 90

Pixel Values with Average or Message Characters
251 253 (251+67)/2 = 159
252 249 (233+104)/2 = 168
243 232 (187+97)/2 = 142
235 215 (142+112)/2 = 127
233 208 (124+116)/2 = 120
248 222 (128+101)/2 = 114
239 218 (107+114)/2 = 110
238 221 (98+32)/2 = 65
232 212 (90+49)/2 = 69
```

The slightly improved image would look like the one here.
A moderate improvement!

Note that in this case, you will need the original image in order to extract the original message because the information from the image is mixed with the information of the message.

The PPM format

We have worked with PPM formatted images before, namely in CSC226 (**http://cs.berea.edu/courses/csc226/tasks/L3.ppm.html**). This lab is a variation of that assignment, so it would be useful to read it to refresh your memory of what we did in that assignment.

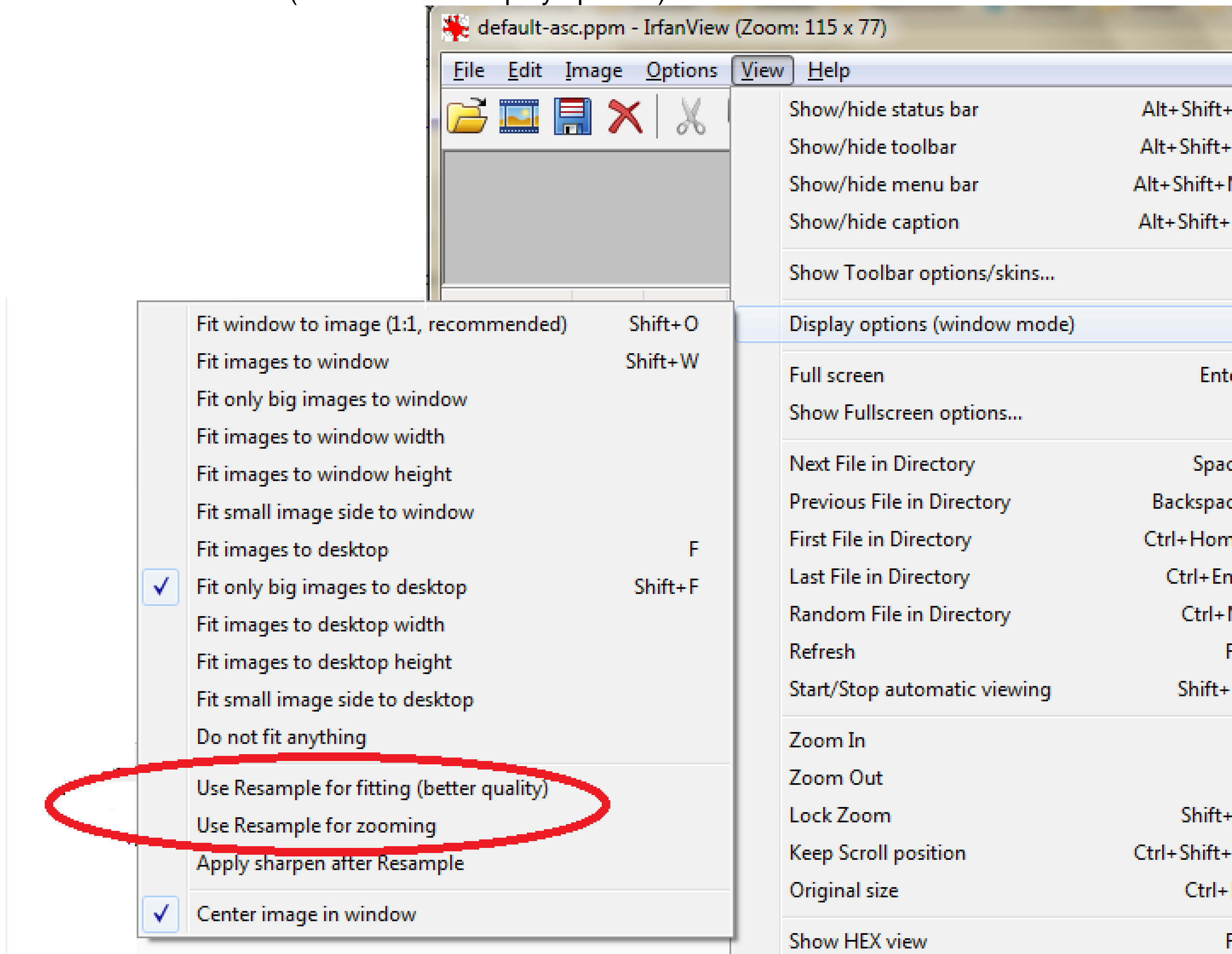
We will highlight some concepts here.

Some of the files we used in that assignment that are relevant in this lab are:

1. **ppm.py**
   A module defining the PPM class we created which has methods for loading, displaying and saving PPM images. It is in this module that you can add methods that will take a message and embed it into the image. In order to make the embedded image less corrupted looking, you must integrate the message information with the information in the image. You must use this class, though you may modify it if desired.
2. **yourusername-L3-ppm.py**
   The "driver" module that imported the PPM class and created an instance of that class that the user manipulated. If you choose to reuse this file, you can put the code to interact with the user for the image that is to be loaded and the message to embed into the image.
3. **bc-sign.ppm**
   The bc-sign image in PPM format. Recall that an image in PPM format is easy to view in a text editor like Notepad in which the saturation values for each pixel is listed out as a trio of numbers that go from 0 to the max value stored in the header. Because the image has 25,000 pixels, this file contains 75,000 integers for the image in addition to the header information!

Go ahead an open the PPM file and see what we mean.

You can use this file to test your program, but remember that you can also convert pictures in various formats into PPM using **irfanview** if you have another favorite image that you prefer Just remember that you want to make these images SMALL because these files are uncompressed, and you want to use the PPM P3 format so it is in ASCII. Note that if you use extremely small images and want to see pixels, we recommend turning the resampling settings off in IrfanView (under view -> display options.)



Impact of memory management on the PPM class

Note that because you will need to extract the message from the embedded image to check to make sure it works, you will probably need to keep a copy of the original image, maybe as an instance variable of the PPM class if that is what you choose. This requirement presents an interesting challenge due to the way that Python manages lists in memory.

Recall that the PPM class loads an image in PPM format into a list of rows of the image, and each row is a list of pixels for that particular row. Additionally, a pixel is represented as a list of three integers, one for each of the RGB colors. Therefore, a color image is a list of a list of a list of integers, which makes it an issue if you are trying to make a copy of the image.

If you do not make a deep copy of the original image, any changes you make when integrating the message into the embedded copy can potentially modify the original image. Think for a minute what that would mean.

Both the original and embedded images would be identical; would that be a problem?

Lab Specifics

You want to send a message to a friend via email, but you are not comfortable that your email is completely secure. Therefore, you decide to design and create a program that will take an image and a text message stored in two separate files and embed the message into the image, which you then send as an email attachment.

You realize that you need to include an ability for your program to extract the message back from the image to make sure that your algorithm was successful. Therefore, your final program would be able to both encode an image with text that you get from a file and to be able to extract that message back. You can assume that your friend has the original image that she will use to extract your message.

You will need:

1. a file called *yourusername(s)-L01-message.txt* to store the message you wish to embed into the image.
2. a file called *yourusername(s)-L01-image.ppm* that contains the image you will use to hide the message.
3. the **ppm.py** module that you modified so that the PPM class can embed the message into the image as well as extract it back out.
   Make sure you include appropriate docstrings for all methods and informative comments for ALL changes you make to this file.
4. a driver module named *yourusername(s)-L01-driver.py* that contains the code for your program to interact with the user. Whether or not you choose to download or modify **yourusername-L3-ppm.py** or create a new module, you must include comments at the top with information about the author(s) and what your program is doing.
   You should use this format:
5. `##   Lab L01: Steganography`

```
6. ##   Your names: put your names here
7. ##   Contributions: list each person's name and their specific
   contribution (navigator, driver)
8. ##   Purpose: put what your program does here
   ##   Acknowledgments: put any acknowledgements here
```

You should also include informative comments in your code and appropriate docstrings if you defined any new functions in this module.

Reflection

You are work with a partner for this lab or individually.
If you decide to work with a partner, be sure to follow good "**pair-programming**" practices as you did for CSC226.

Answer the following questions in *yourusername(s)-L01-reflection.docx*:

- AUTHORSHIP: Describe who did the work on this lab. If you worked as a pair, did you use good pair-programming practices? Explain.
- INITIAL DESIGN PLAN: What is a pseudocode design plan which meets the computational requirements of this lab?
- SUMMARY: A brief summary description of the design and implementation, including how much your initial design plan evolved, the final result you achieved and the amount of time you spent as a programmer in accomplishing these results. This should be no more than two paragraphs.
- IMPLEMENTATION: A list in bullet form of specifically what was accomplished including any challenges overcome and innovations that were not specifically required by the assignment.
- TESTING: A list in bulleted form of all input values used for testing. Here you should be careful to select representative input cases, including both representative typical cases as well as extreme cases.
- FILES: A list in bulleted form of the names of all files submitted (source code and input, etc.)
- ERRORS: A list in bulleted form of all known errors and deficiencies.
- COMMENTS: A paragraph or so of your own comments on and reactions to the Lab.
- BIG-O on embed: What is the big-O analysis of the method that embeds the message into your image? You may assume that the image is dimension nxm and the message is of size x. Be careful to consider the situation when the message has more characters than there are pixels in the images, or if it has fewer.
- BIG-O on extract: What is the big-O analysis of the method that extracts the message from the image? You may assume that the image is dimension nxm and the message is of size x.

Submission

1. Create a folder called *yourusername(s)-L01*
2. Copy *yourusername(s)-L01-message.txt* into it.
3. Copy *yourusername(s)-L01-image.ppm* into it.

4. Copy your modified *ppm.py* module into it.
5. Copy *yourusername(s)-L01-driver.py* into it.
6. Copy *yourusername(s)-L01-reflection.docx* into it.
7. Zip up this directory into a file called *yourusername(s)-L01.zip*
8. Submit this file to Moodle. If you are pair-programming, one person should submit this file and the other person will write the name of the partner in their submission.

Copyright 2016 | Licensed under a Creative Commons Attribution-Share Alike 3.0 United States License | http://cs.berea.edu/CSC236/