CISC-121 Winter 2013

Weeks 11 and 12 Lab Assignment: Your Choice

This is the assignment for the labs in Weeks 11 and 12 (March 21 – April 5). **This assignment will be graded.**

Deadline:

Please submit your solution by 12:00 Midnight Monday April 8. I will give you a 24-hour grace period that ends at 12:00 PM Midnight Tuesday April 9. Assignments submitted during the grace period will be penalized 10% for lateness. Assignments will not be accepted after the end of the grace period. You may work in pairs on this assignment.

This assignment gives you the choice of two problems to work on. The first is an image-manipulation activity, and the second involves a variation on the standard binary search tree. You ARE NOT required to submit solutions for both problems, but I encourage you to at least try to solve them both, as programming practice.

OPTION 1

Introduction:

There are two broad categories of protecting confidential information and communication: cryptography – the encoding of information so that it can be seen but not understood – and steganography – the concealing of information so that its presence is undetected. Modern cryptography tends to rely on complex mathematical operations, whereas steganography, both ancient and modern, is often based on creative craftiness. In this assignment you will develop a Python program to perform a simple form of steganography – hiding text messages inside images**.** As with the previous hand-in assignment, this assignment requires the pypng library.

Background:

According to an ancient story told by the Greek historian Herodotus, a message was once tattooed on a messenger's shaved head – delivery was delayed until the hair grew back to cover the message. More practical forms of steganography such as invisible ink have been used for millennia. In modern times, messages have been photoreduced and hidden in punctuation marks (the fabled "microdots"), or in excess whitespace between lines of text.

With digital imagery, the task of steganography has become much easier. There are many ways of hiding text inside images – we will look at a very simple one.

As we know, each pixel in a PNG image is represented by 3 values: a red value, a green value, and a blue value, each of which can take on values in the range [0 ... 255].

Coincidentally, each ASCII character has a corresponding integer with no more than 3 digits.

Try executing the following code:

```
for c in "Woot!":
    print c," : ",ord(c)

for n in range(90,110):
    print n," : ",chr(n)
```

to see examples of this.

The average human eye cannot actually distinguish between colours that have very similar pixel colour values. To a human observer, changing the colour values for a pixel by a small amount does not create a noticeable difference.

This means we can hide an ASCII character in a pixel: convert the character to its equivalent 3-digit integer, then hide the first digit in the red value, the second digit in the green value, and the third digit in the blue value. In the case of an integer less than 100, the first digit is 0. In the case of an integer less than 10, the first and second digits are both 0.

For example, suppose the pixel has colour values [83,234,7] and we wish to hide an "m" in this pixel. We find that ord("m") is 109. We change the **last digits of the pixel values** to match the digits of 109 - giving [81,230,9], which is visually very close to the original pixel. Later, to extract the hidden letter we take the last digit of each pixel value – 1, 0, 9 – and put them together to make 109. Chr(109) will give us back the desired letter "m".

There is a small but important point. It is not sufficient to simply replace the last digit of a pixel's colour value by the desired digit from ord( ) - for example if the colour value is 255 and we want to convert the last digit to 9 ... we must change the colour value to 249 rather than 259 because 259 is not a valid colour value. Also, in general we try to minimize the change to a colour value – so for example if we want to change the last digit of colour value 220 to 9, we should change it to 219 rather than 229, because 219 is closer to the original value.

END OF MESSAGE marker: in general the hidden text message will occupy only a few of the pixels in an image (unless you are hiding the text of War and Peace). The pixels that follow the end of the message will produce nonsense or non-characters when decoded. To avoid this, the pixel immediately following the last character of the message should have the number 257 (or some other non-character number) hidden in it. Your extraction function should check for this value each time before it attempts to get a character out of a pixel.

Starting Point:

You are provided with a stub of a program. The provided program contains a menu with three options: hide a message in an image, extract a message from an image, and exit the program.

Requirements:

A: Your first task is to write the two functions corresponding to the hiding and extracting operations.

B: Detection Avoidance: One of the problems with this form of steganography is that it is quite detectable. One method of testing an image to see if it might contain a message is to

look at the last digits of the colour values for the pixels at the beginning of the file. In an undoctored image, these values should be fairly uniformly distributed. But if the beginning of an image file has (for example) a lot of pixels whose colour values end in 1,0,1, it is a reasonable guess that those pixels are all hiding the letter “e” .

Your second task is to enhance your hide_message() function to defeat detection by the kind of frequency analysis just mentioned. There are numerous ways to achieve this. One simple way is to spread the text out across more of the image by leaving groups of untouched pixels between the ones where you hide the characters. Here are two possibilities - you may choose one of these or design your own, based on these or other ideas:

- In the first 2 pixels, hide a 2 digit number. After that, skip over that many pixels before hiding the next character of the text.
- After hiding the first character in the first pixel, use the average of the colour values for this pixel to determine how many pixels to jump over before hiding the next character. Then use the average of that pixel's colour values to determine how far to jump before hiding the next character, and so on.

C: Enhanced Extraction: Whichever method you used in B, modify your extract_text() function so that it extracts a message hidden using your enhanced technique.

Sample Images and Text:

I have provided a few images and text files that you may use for developing and testing your functions. Feel free to use your own images or text files if you wish.

Dealing with Errors:

Your program and the functions you write should be robust. You can assume that any file that ends in “.png” does contain a valid PNG image. Your program should handle a situation where the image is not big enough to hide the entire message.

Marking Scheme:

This assignment will be marked out of 100:

1. Correctness : 70
2. Error Handling : 15
3. Style and documentation : 15

Documentation:

1. At the beginning of your program, identify yourself, your working-partner (if any), and the date on which you are submitting your solution
2. Identify the purpose of each function
3. Add comments where ever you feel they will help the reader understand your algorithm.

Uploading Your Solution:

We will use Moodle to collect your solutions, as usual.

OPTION 2

Introduction:

In class we have seen the basic operations on a sorted binary tree: search, insertion, and deletion. In this assignment you are asked to build a special type of sorted binary tree: one which attempts to reduce search time by placing values closer to the root if they are more likely to be searched for.

As a simple example, suppose our tree contains only the words "apple" and "pie". If we know that "pie" is more likely to be searched for than "apple", it makes sense to put "pie" in the root of the tree.

It is possible to create an "optimal" binary search tree which minimizes the average search time, but the algorithm for this is beyond the scope of CISC-121. Instead, we will approximate it by requiring that in every subtree, the most-frequently-searched-for object is at the top. This type of tree is called a max-heap, and is extremely useful in many applications.

Building a sorted binary tree with the max-heap property is difficult if the contents of the tree and the search-frequencies change over time, but it is really easy if we have all the data and the search frequencies – we just sort the data so that the most-frequently-searched-for item is first and the least-frequently-searched-for item is last, then build the binary tree – the resulting binary tree will have the desired property.

In this assignment you get to take the easy route. Your assignment is to build two trees containing the same data:

- one tree by adding the items in an arbitrary order – we will use the order of the items in a Python dictionary
- one tree by adding the items in order of frequency

and then do some tests to see if the frequency-based tree is more efficient in terms of search times.

Starting Point:

You are provided with a Tree Class and a Node Class. There are also some sample text files in the zipped folder.

Requirements:

Write a main program that

- asks the user for the name of a text file, then reads in the file and separates it into words.
- eliminates all punctuation and converts all words to lower case
- creates a dictionary in which the words are the keys and the stored value for each word is the number of times it occurs in the text file. For example, if the text file contains the word "candle" 17 times, then your dictionary should end up with the pair **"candle" : 17** stored in it. We will use these numbers to represent the search-frequency of the words
- iterates through the dictionary, adding the words to a binary search tree. Since the dictionary is not stored in a predictable order, this is an approximation to adding the nodes in a random order
- creates a list of the words, sorted according to their frequency in the text file. You may use any sorting method you choose (this is the only time I will ever give permission to use an O(n^2) sort such as bubble-sort).
- iterates through the list, adding the words to another binary search tree – this one will have the max-heap property.
- determines the height of each of the two trees you have built. The height of a tree is the maximum number of steps from the root to any other node in the tree. This can easily be done recursively: the height of a tree at each node is 1 + the larger of the heights of the node's children. In general, a tree with lower height will have lower average search-time than a tree with greater height
- selects a large (you decide how many) number of words and searches each tree for them. Some of your searches should be for words that are not in the trees. For each search you need to keep track of the number of nodes that were examined during the search – this will require modifying the existing search function or creating your own, so that it returns the number of nodes examined

Once you have created your program, conduct experiments to explore the average search time (as measured by the number of nodes examined during each search) for the trees you have built. Experiment on at least 5 different text-files (I have provided you with 2 small ones). For each file

- compare the height of the two trees
- search both trees for particular words chosen from the text-file. Search words should be chosen so that the probability of a word being chosen as a search-word is determined by its frequency in the file. The easy way to do this is create a list from the words in the text file where each each word is repeated in the list as many times as it is repeated in the text file. Then randomly choosing a word in the list will favour the words that occur most frequently in the file.

- search both trees for words chosen completely at random from a different text-file

In your submission, include a discussion of your experiments, results and conclusions at the end of your program. This discussion should be enclosed in """ """ delimiters. Since the programming in this option is minimal, the quality of your experiments and discussion will be worth 35% of the marks.

Marking Scheme:

This assignment will be marked out of 100:

1. Correctness : 35
2. Discussion: 35
3. Error Handling : 15
4. Style and documentation : 15

Documentation:

4. At the beginning of your program, identify yourself, your working-partner (if any), and the date on which you are submitting your solution
5. Identify the purpose of each function
6. Add comments where ever you feel they will help the reader understand your algorithm.

Uploading Your Solution:

We will use Moodle to collect your solutions.

Academic Integrity:

You must observe the Queen's Academic Integrity Policy in this assignment. All of the work you submit must be your own work (or the work of you and your partner, if you work in a pair). You may **not** copy from **any source –** the Internet, a book, or any other source. Even if you cite your source, this is not acceptable: you should be learning how to write your own code. The TA's and I will give advice but we will not write your code for you. You may discuss the assignment with other students but you must not look at or exchange code or even pseudo-code with each other (except your working-partner, if any). If we discover two solutions that are too similar to be a coincidence, you will be charged with a departure from academic integrity. Queen's has a zero-tolerance policy in such cases.