Foundations of Computing (COMP10001)
Semester 2 2013
Project 2 Specification

Steganography

Version 2, 16 September 2013

Changes from Version 1:

- Changed the parameter name of the `bits_to_message` function in Task 2 from "`bits`" to "`message_bits`" to avoid shadowing the "`bits`" module name.

Introduction

Alice and Bob are friends who communicate frequently via email. Recent stories in the news about the lack of email privacy have made them nervous that someone else might be reading their correspondence. They decide to hide their messages inside digital images (computer graphics files) to make it harder for an interceptor to spy on them. This is an example of *steganography*, where a message is concealed inside another piece of data. If a spy manages to read the email between Alice and Bob they will see innocent-looking images being sent back and forth, but they will be oblivious to the secrets hidden within those images. Of course Alice and Bob can share their encoded images in other ways – posting them on a web page would work just as well.

In this project you will implement a Python program to help Alice and Bob use steganography to maintain their privacy. In particular you will write a program which can *encode* a piece of text inside an existing image, and also *decode* a piece of text from a previously encoded image.

Important background information

Alice and Bob agree to use the ASCII character set for their secret messages. There are 256 characters in the ASCII set, and each character has a corresponding numerical code in the range $[0, 255]$ – this is the number returned by the `ord` function in Python. If we write the character codes in base 2 (binary) then we only need a maximum of 8 binary digits (bits) to represent the whole set, because $2^8 = 256$.

We generally assume that numbers are written in base 10. However, the choice of base is quite arbitrary, and we could just as well use another base instead. When we are being explicit about the choice of base it is conventional to write it as a subscript of the number. For example 165_{10} means $\mathbf{1} \times 10^2 + \mathbf{6} \times 10^1 + \mathbf{5} \times 10^0$. However, we can write *the same number* in base 2:

$$\begin{aligned} 10100101_2 &= \mathbf{1} \times 2^7 + \mathbf{0} \times 2^6 + \mathbf{1} \times 2^5 + \mathbf{0} \times 2^4 + \mathbf{0} \times 2^3 + \mathbf{1} \times 2^2 + \mathbf{0} \times 2^1 + \mathbf{1} \times 2^0 \\ &= 128 + 0 + 32 + 0 + 0 + 4 + 0 + 1 \\ &= 165_{10} \end{aligned}$$

A key part of Alice and Bob's steganography scheme is to convert the secret message into one long string of bits. This is done by converting each character into its ASCII code, then writing each code in binary, then appending them all together in sequence. To make it easy to decode, each binary number is written in exactly 8 bits, with zeros padding on the left where necessary. Consider, for example, the binary encoding of the message `"hello"`. First, each character is converted into its corresponding ASCII code, which gives us the numbers 104_{10}, 101_{10}, 108_{10}, 108_{10}, and 111_{10}. Second, each code is written in base 2: $104_{10} = 01101000_2$, $101_{10} = 01100101_2$, $108_{10} = 01101100_2$, $108_{10} = 01101100_2$ and $111_{10} = 01101111_2$ (note carefully that we've written each binary number with exactly 8 bits). Finally all the binary numbers are appended together to give one long sequence of 40 bits:

0110100001100101011011000110110001101111

How will this message be hidden inside an image? To answer that question we need to know how images are represented. For the purposes of this project, images will be represented in Python as rectangular two-dimensional list of pixels, in zero-based, row-major order (similar to the grid in the first project). The top-left pixel is at coordinate $(0, 0)$ and the bottom right pixel is at $(r-1, c-1)$ in an image with r rows and c columns. Each pixel specifies a single dot in the image, and is represented in Python by a three-element tuple of integers, encoding the *red*, *green* and *blue* component colours of the dot, in that order. Each integer specifies the *intensity* of the corresponding component colour within the range $[0, 255]$.[1]

Below is an example 3×2 image in Python (written over multiple lines to make it easier to read):

```
image = [
    [(15, 103, 255), (0, 3, 19)  ],
    [(22, 200, 1),   (8, 8, 8)   ],
    [(0,  0,   0),   (5, 123, 19)]
 ]
```

The pixel at coordinate $(2, 1)$ is accessed via `image[2][1]`, yielding the tuple `(5, 123, 19)`, with a red intensity of 5, a green intensity of 123 and a blue intensity of 19.

The rightmost digit in a number is called the *least significant digit* because it contributes the least amount to the overall value. For example, the least significant digit of 123_{10} is 3, whereas the least significant digit of 124_{10} is 4. When we are talking about binary numbers it is common to refer to the least significant digit as the *least significant bit* (LSB). Therefore the LSB of 1111011_2 is 1, and the LSB of 1111100_2 is 0. If you imagine a pixel intensity value in base 2, the LSB contributes only a tiny amount to the intensity of the corresponding component colour; only 1 intensity level out of 256. In many cases, a casual observer cannot tell the difference between two intensity values which differ by only 1. Consequently we can change the LSBs of the pixel intensities in an image without having a noticeable effect on its visual appearance; Alice and Bob's steganography scheme takes full advantage of this fact. The idea is to take each bit in the message and set the LSB of an intensity value in the image to be equal to that bit, thus embedding the message in the pixel data.

Obviously the order of the bits in the message is important, so we need to decide which bits from the message will be set in which intensity values in the image. There are three intensity values per pixel, which complicates matters somewhat, but the basic idea is to associate triplets of bits from the message with pixels visited in order from left-to-right and top-to-bottom. In other words, we start with the first row of the image, and visit each pixel in that row from left-to-right, then we move to the next row and repeat until we've visited the whole image. The first bit in a triplet of consecutive bits is associated with the red intensity value, the second bit with green and the third bit with blue. The table below shows how the first 18 bits from the example message are associated with each of the intensity values from the example image. Thick vertical lines show how triplets of bits are associated with each of the red, green and blue components of each pixel. Note carefully the order in which the intensity values are taken from the image.

message bit	0	1	1	0	1	0	0	0	0	1	1	0	0	1	0	1	0	1
intensity	15	103	255	0	3	19	22	200	1	8	8	8	0	0	0	5	123	19

The astute reader will have noticed that there are $5 \times 8 = 40$ bits in the binary message, but there are only $2 \times 3 \times 3 = 18$ intensity values in the image. In this case, the example image is too small to encode the entire message – we only have enough intensity values to encode the first two characters of the message (`he`), and we have two spare message bits left over. It is also possible that a message will contain fewer bits than there are intensity values in a given image. Alice and Bob decide that if the image is too small to contain the entire message they will encode as many bits from the message as possible, even if that number is not a multiple of 8. If the message has fewer bits than the image has intensity values, they will set the LSBs of the left over intensity values to 0. They are careful to decode an image in chunks of 8, to make sure they only consider complete characters, and they will stop decoding if they encounter a character represented by 8 zero bits in a row (00000000).

The table below shows how the first 18 bits of the message are embedded in the example image. The first two columns show the original intensities in base 10 and base 2. The third column shows the

[1] It is a coincidence that both the ASCII codes and the pixel intensity values are both integers in the range $[0, 255]$.

corresponding bit from the message. The fourth and fifth columns show the new *encoded* intensities, which are computed by setting the LSB of the original value to the associated bit. Thick horizontal lines show how triplets of bits are associated with each of the red, green and blue components of each pixel.

original intensity in base 10	original intensity in base 2	message bit	encoded intensity in base 2	encoded intensity in base 10
15	00001111	0	00001110	14
103	01100111	1	01100111	103
255	11111111	1	11111111	255
0	00000000	0	00000000	0
3	00000011	1	00000011	3
19	00010011	0	00010010	18
22	00010110	0	00010110	22
200	11001000	0	11001000	200
1	00000001	0	00000000	0
8	00001000	1	00001001	9
8	00001000	1	00001001	9
8	00001000	0	00001000	8
0	00000000	0	00000000	0
0	00000000	1	00000001	1
0	00000000	0	00000000	0
5	00000101	1	00000101	5
123	01111011	0	01111010	122
19	00010011	1	00010011	19

There are a couple of things to note about the above table:

1. The LSB of the encoded intensity is always equal to the corresponding message bit.
2. The original intensity is different to the encoded intensity (only) when the LSB of the original intensity is different to the corresponding message bit.

Below is the encoded image from the above table in Python syntax, with the intensities in base 10:

```
[[(14, 103, 255), (0, 3, 18)  ],
 [(22, 200, 0),   (9, 9, 8)   ],
 [(0,  1,   0),   (5, 122, 19)]]
```

Compare this to the original image to see how they differ.

Alice could send this encoded image to Bob and he could decode it by inspecting the LSBs of the pixel intensities. As already noted, the image is too small to contain the entire message, so he will only be able to retrieve the text "`he`". Obviously in practice they will use larger images to encode much longer messages.

For Alice and Bob's steganography scheme to work it is essential that the intensity values written to an image file are exactly the same as the intensity values you get back when the file is read in again. If not, the decoded message might be different than what was encoded. When saving their files they must use an image format which is *lossless* – one that does not lose (or alter) any of the intensity values. Not all image file formats have this property. For example the popular JPEG (Joint Photographic Experts Group) image format uses a *lossy* data compression technique to reduce the storage space required by files. The intensity values written to a JPEG file may not be exactly the same as the intensity values that are returned when the file is read again. Fortunately there is another popular image file format called PNG (Portable Network Graphics) which is lossless, so Alice and Bob agree to use that format for their files. By convention PNG file names typically have a `.png` extension, such as `floyd.png`.

Support code

To help you complete the project and test it on real images we have provided two libraries of Python code:

1. `bits.py`: for bit manipulation.
2. `SimpleImage.py`: for reading and writing image files.

You can find them linked project page on LMS. You will need to upload them to your IVLE account. You should read the comments in those files to see what functionality they provide and how to use them.

Programming tasks

To complete this project you are required to submit a single Python module called `proj2.py` which implements (at least) the four functions described below. You must name those functions as specified in this document, remembering that Python is case-sensitive. Failure to name your module or functions correctly will interfere with our automated testing system and result in a marking penalty. You may also include as many helper functions in your project that you feel are necessary. The marking scheme is covered in more detail at the end of this document.

Task 1 (2 marks)

Write a Python function `message_to_bits(message)` which takes a string of ASCII characters as its argument and returns a string of binary digits (`'1's` or `'0's`) as its result. Each character in the input string should be converted to the base 2 representation of its ASCII code, and the binary codes should be appended together in order of the characters in the message. The base 2 representation of each character's code should be *exactly* 8 characters long. You may find the function `char_to_bits` in the `bits.py` library useful for this task. Below are some sample uses of the `message_to_bits` function at the Python console to illustrate its intended behaviour:

```
>>> message_to_bits('')
''
>>> message_to_bits('A')
'01000001'
>>> message_to_bits('hello')
'0110100001100101011011000110110001101111'
>>> message_to_bits("ABC\n123")
'010000010100001001000011000010100011000100110010001100011'
>>> message_to_bits("je t'aime")
'011010100110010100100000011101000010011101100001011010010110110101100101'
```

To use the functions in `bits.py` you should import it at the top of your own program like so:

```
import bits
```

When you call the functions from that library make sure you prefix their names with the `bits` module name, such as `bits.char_to_bits(...)` and so forth (where `...` is replaced by the actual argument you want to pass to the function).

Task 2 (3 marks)

Write a Python function `bits_to_message(message_bits)` which takes a string of binary digits (`'1's` or `'0's`) as input and returns a string of ASCII characters as its result. You may assume that the input string will not contain any characters other than `'1'` and `'0'`. The input string should be processed 8 bits at a time. Each 8 bit chunk should be converted to an integer (in the range $[0, 255]$) and then that integer should be converted to the corresponding ASCII character. Note the input list of bits may not necessarily have a length which is a multiple of 8. The function should stop processing the input string when either:

- There are fewer than 8 bits left.
- It decodes a character with the ASCII code of `0` (which would be 8 consecutive `0` bits, starting at an index which is a multiple of 8). This sequence of bits marks the end of the encoded message, but its corresponding ASCII character should not be included in the output message itself.

You may find the function `bits_to_char` in the `bits.py` library useful for this task. Below are some sample uses of the `bits_to_message` function at the Python console to illustrate its intended behaviour:

```
>>> bits_to_message('')
''
>>> bits_to_message('0100000')
''
>>> bits_to_message('01000001')
'A'
bits_to_message('011010000110010101')
'he'
>>> bits_to_message('0110100001100101011011000110110001101111')
'hello'
>>> bits_to_message('011010000110010101101100000000000110110001101111')
'hel'
>>> bits_to_message('000000000110100001100101011011000110110001101111')
''
>>> bits_to_message('010000010100001001000011000010100011000100110010001100011')
'ABC\n123'
```

A useful way (but not the only way) to test your solutions to the first two tasks is to convert a message to bits and then back to a message again, and check if the output is equal to the input. The following function should return `True` for all valid input messages:

```
def round_trip(message):
    return bits_to_message(message_to_bits(message)) == message
```

Task 3 (5 marks)

Write a Python function `encode(image, message)` which takes an image as its first argument, a message as its second argument, and returns an encoded image as its result. The encoded image should have the input message embedded into its pixel values following the steganography scheme used by Alice and Bob. The input image is in the format described earlier, which is a rectangular two-dimensional lists of pixels, in zero-based, row-major order. Each pixel is a three-element tuple of integers in the range $[0, 255]$. The input message is a string of ASCII characters. To implement this function you should first convert the input message into a binary string using the `message_to_bits` function from Task 1. Then you should iterate over each pixel in the image, starting with the first row, from left-to-right, and set the LSB of each intensity value to the corresponding bit from the binary string. Once the first row of the image is encoded, you should move onto the second row, and so on. You may find the function `set_bit` from `bits.py` useful for this task. If there are more bits in the binary string than intensity values in the image, your function should encode as many bits of the binary string as possible, even if it is not a multiple of 8. If there are fewer bits in the binary message than intensity values in the image, your function should set the LSBs of any remaining intensity values to 0. Below are some sample uses of the `encode` function at the Python console to illustrate its intended behaviour:

```
>>> test_image = [[(15, 103, 255), (0, 3, 19)],
                  [(22, 200, 1), (8, 8, 8)],
                  [(0, 0, 0), (5, 123, 19)]]
>>> encode(test_image, "hello")
[[(14, 103, 255), (0, 3, 18)], [(22, 200, 0), (9, 9, 8)], [(0, 1, 0), (5, 122, 19)]]
```

```
>>> encode([], "hello")
[]
>>> encode(test_image, "")
[[(14, 102, 254), (0, 2, 18)], [(22, 200, 0), (8, 8, 8)], [(0, 0, 0), (4, 122, 18)]]
```

Note: the first statement above has been written over multiple lines to make it fit on the page, but you can write it all on one line in IVLE.

You will probably want to test your `encode` function on larger images. To make it easier to do this we have provided functions in the `SimpleImage.py` library called `read_image` and `write_image` to read and write images to and from PNG files. Supposing you have a file called `floyd.png` in your IVLE account, you can use the functions like so:

```
>>> from SimpleImage import read_image, write_image
>>> floyd_image = read_image('floyd.png')
>>> floyd_image_encoded = encode(floyd_image, 'Floyd is cute!')
>>> write_image(floyd_image_encoded, 'floyd_encoded.png')
```

Assuming you implemented your `encode` function correctly, the above statements should create a file called `floyd_encoded.png` in your IVLE account, which has the message "`Floyd is cute!`" embedded within it. If you view the files `floyd.png` and `floyd_encoded.png` side-by-side it should be very difficult to tell them apart.

We provide some example PNG files to get you started on the project page on the LMS, you should upload them to your IVLE account.

Task 4 (3 marks)

Write a function `decode(image)` which takes an image as its argument and returns a string of ASCII characters as its result. The output string is the result of decoding the message by following Alice and Bob's steganography scheme described above. The input image is in the same format as described in Task 3. To implement this function you should collect the LSB from every intensity value in the image in the same order that the bits were encoded, producing a string of bits. Then convert this string of bits to the resulting string of ASCII characters using the `bits_to_message` function from Task 2.

Below are some sample uses of the `encode` function at the Python console to illustrate its intended behaviour:

```
>>> test_image = [[(15, 103, 255), (0, 3, 19)],
                  [(22, 200, 1), (8, 8, 8)],
                  [(0, 0, 0), (5, 123, 19)]]
>>> decode(encode(test_image, ''))
''
>>> decode(encode(test_image, 'hi'))
'hi'
>>> decode(encode(test_image, 'hello'))
'he'
```

If you followed the steps in Task 3 to produce the file `floyd_encoded.png` you can decode it like so:

```
>>> from SimpleImage import read_image
>>> decode(read_image('floyd_encoded.png'))
'Floyd is cute!'
```

Bonus Task (up to 2 bonus marks)

If you complete the program early and are looking for more fun things to do, you might want to consider generalising Alice and Bob's steganography scheme so that it can encode more than one bit from the message in each intensity value in the image. It ought to be possible to encode up to 8 bits of the message per intensity value. More bits per intensity value means longer messages can be encoded in the same image size. Of course the downside is that the encoded image will begin to look

less like the original as more and more bits are used per intensity value, and may look more suspicious to someone spying on their email.

Write two functions:

```
encode_ext(image, message, num_bits)
decode_ext(image, num_bits)
```

which work in a similar fashion to the functions from Tasks 3 and 4, but take an extra integer parameter called `num_bits` in the range $[1, 8]$ which specify how many bits of the intensity values have been used to encode bits from the message. It is up to you to decide how the bits from the message are encoded into the pixel intensities, as long as your encoded messages can be correctly decoded.

If you attempt this bonus task make sure your submitted program includes all the functions required in Tasks 1-4.

Task 5: Programming Style (2 marks)

Your submitted code should follow the principles of good programming style set out in the *COMP10001 Programming Style Guide*, which can be downloaded from the same location as this project specification on LMS. Your submitted code should be properly documented with appropriate comments, including:

- Module header comments (at the start of the file):
  - A description of the program and its purpose.
  - Your name and the names of anyone who has contributed directly to the code.
  - The date the module was first created.
  - A dated list of modification notes, which explain the significant changes you have made to the program as it is developed.
- Comments for significant functions and variables:
  - An explanation of what the variable/function is for.
  - An explanation of the arguments and results of functions.
- In-line comments which explain the key parts of the implemented algorithms.

Marking scheme and project submission

The project is worth 15 marks (15% of your final mark for the subject).

Marks may be deducted for the following reasons:

- Incorrect names for one or more of the required functions: 1 mark deducted.
- Incorrect name for the submitted Python module: 1 mark deducted.

Up to 2 bonus marks may be awarded for solutions to the bonus task, but your total mark for the project will be capped at 15.

This project should be completed on an *individual* basis. While it is acceptable to discuss the project with others in general terms, excessive collaboration and any sharing of code are considered a breach of the University's Academic Misconduct policy (`http://academichonesty.unimelb.edu.au/policy.html`). Submission of your project constitutes a declaration of academic honesty and that your submission is strictly your own work (with the exception of the library code made available to you).

Submitting your project

Submission will be via IVLE, in the same way that Project 1 was submitted (but remember to submit to `proj2`). If you forget how to submit your project, please consult the documentation provided in Project 1 for submitting projects on IVLE.

Late submissions will not be accepted other than on documented medical or personal grounds. If such circumstances have taken time away from your project work, you should contact Bernie Pope via email (`bjpope@unimelb.edu.au`) as soon as possible.

Be warned that IVLE is often heavily loaded near project deadlines, and unexpected network or system down-time can occur. You should plan ahead to avoid unexpected problems at the last minute. System down-time or failure will generally not be considered as grounds for special consideration.

Changes/Updates to the Project Specifications

Any changes or clarifications to the project specification will be advertised on the LMS.

Important Dates

- Release of Project 2 specification: Friday 13 September, 2013 (5:00pm)
- Deadline for submission of Project 2: Friday 4 October, 2013 (7:00pm)