Steg

Write a Python program that implements the steg algorithms discussed below. This is a group programming assignment (i.e., **only one submission per group is needed**).

Requirements

You are to write **one** program that implements both the bit and byte methods of storing and retrieving the hidden data (more on this below). The sentinel value will be the six bytes `0x0 0xff 0x0 0x0 0xff 0x0` appended to the hidden data. After careful thought, let's agree to store the bit data from left-to-right. Note that you should be able to alter your program at a later time (e.g., during Cyber Storm) to store the bit data from right-to-left.

Please, no GUIs. Make this a command line application without frills that I can execute at the command line as illustrated in the "Usage" section below.

File size vs. sentinel

There are several ways to go about hiding data in other data (let's call this "other data" a *wrapper*). Many files have headers that, if changed, break the file. So the data must be hidden beyond the header if it exists. To do this, a user-specified *offset* can be implemented that specifies a location in the wrapper that is beyond the header. At this point, an internal header can be created that provides information about the file to be covertly stored in the wrapper. Information such as its file size can assist its eventual extraction by providing a stopping point.

Another implementation involves a sentinel value that, once encountered during extraction, specifies the end of the hidden file. You will only implement the sentinel method; therefore, your program will not have to create a header.

Byte vs. bit

Data can be hidden, a single byte at a time, by replacing entire bytes of the wrapper. Note that this may significantly change the wrapper, however, thereby alerting to the fact that it has been modified. Instead, an *interval* may be implemented that describes a constant distance between each byte of the hidden file in the wrapper.

Another method involves placing a byte of the hidden file in the least significant bit (LSB) of a group of eight bytes of the wrapper. For example, storing the byte `0xFF` into the wrapper bytes `0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0` would result in the modified wrapper bytes `0x1 0x1 0x1 0x1 0x1 0x1 0x1 0x1`. This significantly reduces the chance that the hidden file's contents within the wrapper will be noticeable. You will implement both of these methods.

Usage

```
python Steg.py -(sr) -(bB) -o<val> [-i<val>] -w<val> [-h<val>]
 -s      store
 -r      retrieve
 -b      bit mode
 -B      byte mode
 -o<val> set offset to <val> (default is 0)
 -i<val> set interval to <val> (default is 1)
 -w<val> set wrapper file to <val>
 -h<val> set hidden file to <val>
```

Note that generated output is to be directed to `stdout`. In the case of storing, the combined wrapper and hidden file could be written to disk via a redirection with "`> output.jpg`." Suppose we wish to hide `secret.jpg` within `image.jpg`, at an offset of $1,024$, an interval of 256, using the byte method, and save the file to `new.jpg`. The execution would be performed as follows:

```
python Steg.py -s -B -o1024 -i256 -wimage.jpg -hsecret.jpg > new.jpg
```

The retrieval of the hidden file within `new.jpg` and storage to `extracted.jpg` would be performed as follows:

```
python Steg.py -r -B -o1024 -i256 -wnew.jpg > extracted.jpg
```

Using the bit method would be similar, but with the `-b` command line argument instead and without specifying an interval (although one certainly could be specified).

Byte method

For this method you will store and retrieve the data, byte-by-byte, at some specified interval within the wrapper. Furthermore, you will append the sentinel value to the hidden file's data. An optimal interval, I, depends on the size ratio of the hidden file to the wrapper, taking the specified offset and sentinel into consideration. It is computed as follows:

$$I = \left\lfloor \frac{S_w - o}{S_h + s} \right\rfloor$$

Where S_w is the size of the wrapper, o is the specified offset, S_h is the size of the hidden file, s is the sentinel, and $\lfloor\ \rfloor$ is the `floor` function. Note that all values are specified in bytes. Calculating this automatically would mandate a header since we wouldn't be able to calculate it when attempting to extract hidden data, so a manually specified interval is fine. One might think that the sentinel doesn't affect the interval much in the calculation since it is only six bytes; however, the case when the hidden file is very small compared to the wrapper can lead to a large difference between the interval computed when the sentinel is included and when it is not. If this occurs, the interval may be too large and can lead to the dreaded `Segmentation fault`.

Suppose, for example, that we wish to hide `secret.jpg` (15,827 bytes) within `image.jpg` (19,334,874 bytes) with an offset of 1,024 bytes. The optimal interval would be calculated as follows:

$$I = \left\lfloor \frac{19334874 - 1024}{15827 + 6} \right\rfloor = 1,221 \text{ bytes}$$

Storage of the hidden data and sentinel can be done via a simple algorithm. Here's some pseudocode:

```
W <-- wrapper bytes
H <-- hidden bytes

i <-- 0
While i < length(H)
    W[offset] <-- H[i]
    offset += interval
    i += 1
End While

i <-- 0
While i < length(SENTINEL)
    W[offset] <-- SENTINEL[i]
    offset += interval
    i += 1
End While
```

Extraction is done by seeking past the offset in the wrapper and extracting the hidden data, byte-by-byte, at the specified interval, until the sentinel is reached. The final step is to direct the hidden file to `stdout`. Here's some pseudocode:

```
W <-- wrapper bytes
H <-- empty byte array

While offset < length(W)
    b <-- W[offset]

    # check if b matches a sentinel byte
    # if so, we need to check further
    # if not, we can add this byte to H
    # but we may need to add matched partial sentinel bytes first!
    # afterwards...

    H += b
    offset += interval
End While
```

Bit method

For this method, you will store the data, bit-by-bit, in the LSB of bytes within the wrapper. Furthermore, you will append the sentinel value to the hidden file's data. Storing the data bit-by-bit can be done by first iterating over the bytes in the hidden file. For each byte, the most significant bit (MSB) is extracted and stored in the LSB of a byte in the wrapper. The next MSB is then extracted and stored in the LSB of the next byte in the wrapper. This process is continued until the entire byte has been stored. Thus, each byte of the hidden file is stored in eight bytes of the wrapper (at the LSBs). **Note that you are to use bitwise operations to store hidden file data. No bit strings!** How this might work using bitwise operations is not too difficult: we first isolate the 7 MSBs of the wrapper byte, then isolate the MSB of the hidden byte and store it in the LSB of the wrapper byte, and finally shift the hidden byte to the left (to setup the next bit). Here's some pseudocode:

```
W <-- wrapper bytes
H <-- hidden bytes

i <-- 0
While i < length(H)
    For j=0..7
        W[offset] &= 11111110
        W[offset] |= ((H[i] & 10000000) >> 7)
        H[i] <<= 1 # this could result in values greater than 1 byte*
        offset += interval
    End For

    i += 1
End While

i <-- 0
While i < length(SENTINEL)
    For j=0..7
        W[offset] &= 11111110
        W[offset] |= ((SENTINEL[i] & 10000000) >> 7)
        SENTINEL[i] <<= 1 # see note above*
        offset += interval
    End For

    i += 1
End While
```

In Python, shifting a byte to the left may result in a value that is larger than a byte (i.e., greater than 8 bits). To constrain the value to a single byte, the left shift can be rewritten as follows:

```
B = (B << shift) & (2 ** length - 1)
```

Where `B` is the byte in question, `shift` is the number of shifts to perform, and `length` is the number of bits to constrain `B` to. To perform a single shift and constrain the length to 8 bits, for example:

```
B = (B << 1) & (2 ** 8 - 1)
```

Extraction is done by seeking past the offset in the wrapper and extracting the hidden data, bit-by-bit, by rebuilding each byte from the LSBs of eight bytes in the wrapper, until the sentinel is reached. Of course, care must be taken to maintain the interval! **Note that you are to use bitwise operations to retrieve hidden file data. No bit strings!** The final step is to direct the hidden file to `stdout`. Here's some pseudocode:

```
W <-- wrapper bytes
H <-- empty byte array

While offset < length(W)
    b <-- 0

    For j=0..7
        b |= (W[offset] & 00000001)
        If j < 7
            b <<= 1 # see note above*
            offset += interval
        End If
    End For

    # check if b matches a sentinel byte
    # if so, we need to check further
    # if not, we can add this byte to H
    # but we may need to add matched partial sentinel bytes first!
    # afterwards...

    H += b
    offset += interval
End While
```

Hints

- Read the wrapper data from a file as binary data (consider `bytearray`)
- Read the hidden data (if applicable) from a file as binary data (consider `bytearray`)
- Of course, detect if either file is not found and provide an appropriate error message
- You may specify the sentinel as a constant (or variable) at the top of your program (consider storing it as binary data using `bytearray`)
- Write the result to `stdout` as binary data (consider `sys.stdout.buffer.write` in Python 3)

Maths

We can calculate the minimum wrapper size that will contain a hidden file via:

$$S_w >= S_h I + o$$

Note that for the bit method, $I = 8$. Now suppose that we wish to hide `secret.jpg` ($15,827$ bytes) within an image using the byte method with an offset of $1,024$ bytes and a set interval of $1,024$ bytes. The minimum wrapper size is then calculated as:

$$S_w >= (15827)(1024) + 1024 = 16,207,872 \text{ bytes}$$

Of course, the maximum hidden file size that will fit in a wrapper can also be calculated via:

$$S_h <= \left\lfloor \frac{S_w - o}{I} \right\rfloor$$

Using the values specified above (except using the bit method and a wrapper containing $19,334,874$ bytes), the maximum hidden file size is calculated as:

$$S_h <= \left\lfloor \frac{19334874 - 1024}{8} \right\rfloor = 2,416,731 \text{ bytes}$$

One might notice that the sentinel has been excluded from the calculations. Including the six sentinel bytes would not make much difference; however, for posterity:

$$S_h <= \left\lfloor \frac{19334874 - 1024 - 6}{8} \right\rfloor = 2,416,730 \text{ bytes}$$