PSA6: Steganography

Read all instructions in this document before starting any coding!
START EARLY, START OFTEN!

Due: Sunday, November 19th, 11:59pm

Overview: In this assignment you will be embedding secret images in other pictures and also extracting those secret images out from other pictures.

As described in class, **you may complete at most 4 of the quarter's programming assignment using pair programming.** If you have decided to work in pairs, please review the guidelines for pair programming. In addition, note the following details about working with a partner:

- For this assignment, if you decided to work in pairs, find a partner before you start on the assignment. Or else, you have violated the policy on pair programming.
- You will submit only ONE version between the two of you though your group can submit as many times as you want to.

Helpful Information:

Online Communication: Using Piazza, Opening Regrade Requests

Getting Help from Tutor, TA, and Professor's Hours

Lab and Office Hours (Always refer to this calendar before posting.)

Academic Integrity: What You Can and Can't Do in CSE 8A

Remote Lab Access and File Transfer Guide or **Yingjun's SSH and VNC post**

Picture Documentation **and** **Pixel Documentation**

Table of Contents:

Getting Started

Instructions for working on a B230 lab machine:

1) Open a terminal
2) Copy the starter code from the public folder.

Paste the following lines of code one after the other in the terminal i.e. paste the first line, press enter and then the second line and so on:

```
cd ~/
mkdir psa6
cd psa6
cp ../../public/psa6/* .
ls
```

(Do you recall what each of the above commands mean. If you are not able to, review PSA0, where each of the commands is explained.)

There should be 5 pictures (.bmp extension) that should have been copied to your PSA6 folder after executing the above commands. These pictures are from the examples shown in this instruction. More details about what each image represents can be found in Part B of section 5.

3) Verify that the 'PSA6.java' ,'Picture.java' and 'MethodTester.java' files names are shown as part of the result of the last ls command. Following these procedures, your file will be named correctly and be located in the correct places. For every assignment you need to follow correct naming assignments in the correct locations to get credit for your work.

4) Proceed to start your programming assignment.

Instructions for working on your Local Machine (Your Laptop/Computer):

1) Create a psa6 folder on your machine.
2) Download zip and unzip it inside the psa6 folder: LINK HERE

Problem: Steganography (20 points)

Algorithms

This section gives you a high-level view of everything you'll do in this PSA. The next section describes what you will actually do. **But do not skip this section.** It is really important for your overall understanding and your ability to break a problem down into smaller pieces.

Sneaking Information into an Image

Small variations in the red, green and blue values of pixels cannot be discerned by the human eye. For example, the color (39, 56, 100) looks basically the same as the color (42, 55, 102). This information is the key to our ability to hide information inside an image.

Left image is (39,56,100), right image is (42,55,102).

Let's take as a given that we cannot perceive changes of up to plus or minus 3 per color channel in an image. This gives us 2 bits in which to encode any secret information within each color channel. To understand why, let's recall how the numbers 0-255 are represented in binary. You should be very familiar with this by now, but to review, a binary number is a number where each digit can be either 0 or 1, and its location in the number corresponds to a power of 2. E.g., the number **00110100** in binary is the number:

0*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = 0 + 0 + 32 + 16 + 0 + 4 + 0 + 0 = 52

Now, imagine that we change the lowest order two bits at the positions 2^1 and 2^0 (i.e. the rightmost 2 bits). No matter what we set their values to, we can never change the value of this number by more than 3. If we set these bits to 00, the number 00110100 doesn't change. If we set them to 01, it increases by 1. If we set them to 10 it increases by 2, and 11 increases it by 3, but that's all. The same holds for any binary number--no matter what we do to the two lowest order bits, the value of the number will never change by more than 3--thus we will not be able to perceive the color change to the picture.

So, we have 2 bits in each color channel to work with. We can basically set those two bits in each color channel to be anything we want and the viewer of our image won't notice any difference. Obviously we don't have to change those two bits randomly. We want to hide some secret information in those two bits based on the following strategy.

Strategy

Assume we have two pictures named `context` and `message`. `context` is the picture in which we will hide the secret message, and `message` is the picture we will hide. For now assume that the two pictures are of the same size.

The algorithm for hiding `message` inside of `context` is:

1. Make a copy of the `context` image, call it `canvas`.
2. For each pixel in `canvas`:
    a. Find the corresponding pixel in `message` (same x and y position)
    b. For the Red color channel
        i. Project `message` pixel's Red value down onto the values 0, 1, 2, and 3 so that low values map to 0, medium low values map to 1, medium high values map to 2 and high value map to 3.
        ii. Replace the two least significant bits of the `canvas` pixel's Red channel value with the value calculated in step i.
    c. Repeat step b for the Blue channel and the Green channel.
3. Return the modified `canvas` image

This algorithm will successfully hide `message` inside of `canvas`, and you won't be able to see any difference between `canvas` and `context`!

To recover the secret image `message` hidden inside an image `canvas`, we simply do the opposite:

1. Create a new (blank) image called `result` with the same height and width as `canvas`.
2. For each pixel in `canvas`:
    a. Get the corresponding pixel in `result` (same x and y position)
    b. For `canvas` pixel's Red color channel
        i. Get its 2 least significant bits
        ii. Project them up into the range between 0 and 255 (*Do you see why you need to do this? Make sure you do*)
        iii. Set the Red color channel in the `result` pixel to be equal to the value produced in step ii.
    c. Repeat step b for `canvas` pixel's Blue color channel and the Green color channel.
3. Return the `result` picture.

Of course, there are still some details to work out. Keep in mind the abovementioned algorithms are designed for the scenario where `context` and `message` are of the same size. As shown in **Part B of section 5,** this assumption will be relaxed in this PSA. Thus you might need to make some minor changes to the algorithms in this section.

Now, we know what to do, but not necessarily how to do it. For that, read on…

Part A : Writing code - Top-down design (6 points)

In the algorithms above, there are several sub-steps to accomplish. For example, in step 2.b.i in the **hiding algorithm** above, we need to map the 8-bit color channel value down to only 2 bits, preserving as much information as possible.

The method `mostSignificant2(int num)` which you used in **Lab 6** is provided in this PSA's starter code in Picture.java. It can be used to give you the 2 most significant digits of an 8 bit color channel value.

Static Methods

One change you will see between this code and the lab code is that the method is `static` in this PSA. All the methods you write in this PSA **will also be static**.

The major difference between a static method and the methods we have written so far in CSE8A (called instance methods) is ***who*** can use the method. A static method should be used directly from the class, while an instance method has to be called from an object of that class.

For example, the abs method from the `Math` class is a static method. The `getPixels` method from the `Picture` class is an instance method. To call the abs method, we don't have to create a `Math` object; Instead, we can use the `abs` method by invoking it directly from the `Math` class, such as `Math.abs(-5)`. If we want to use the `getPixels` method, we will create a `Picture` object (e.g. `Picture pic=new Picture("swan.jpg");`) and invoke this method from that object (e.g. `Pixel[] pixArray=pic.getPixels()`;).

To determine if a method should be static or instance, we need to think about whether or not the method is related to an individual object of the class. For example, `abs` method's result will be the same no matter which object of the `Math` class calls it (i.e. `abs(-5)` will always be 5 no matter which object invokes this method) . However, the results of the `getPixels` method will be different from different objects (i.e. the returned array will be different from different `Picture` objects). All the methods we write in this PSA are static because their behaviors are not dependent on the objects of the `Picture` class. So, in essence, the methods we write here are one step further in refining our methods from **Lab 6**. We will talk more about static methods later this week in class.

Since we already have a way to get the most significant digits from the secret message to encode, we need to replace the least significant digits in the `context` pixel with these digits.

A.1 Write a method `public static int embedDigits2(int contextVal, int messageVal)` in `Picture.java` (1.5 Points)

- This method will use shift operators (and addition) to replace the two least significant digits in `contextVal` with the value of `messageVal`.
    - You can assume that `contextVal` will always be an integer between 0 and 255 inclusive (i.e. an 8-bit integer) and `messageVal` will always be an integer between 0 and 3 inclusive (i.e. a 2-bit integer).
    - **Hint:** One way to approach this problem is first to "clear" the two least significant digits of `contextVal` (i.e., set them to 0) and then use addition to add in `messageVal`. There are many ways you can clear these two digits, but a quick way is just to shift right and then shift left again. E.g.: 0011 1111 shift right 2 yields 0000 1111. If you shift 0000 1111 left 2 you will get 0011 1100. Voila! Bits cleared.

Some sample outputs from methods calls to **`embedDigits2`** (to fully understand it, write all numbers in binary)..

```
>>> Picture.embedDigits2(182,2)
182

>>> Picture.embedDigits2(179,2)
178

>>> Picture.embedDigits2(166,1)
165

>>> Picture.embedDigits2(12,1)
13
```

A.2 Add another test in MethodTester.java to test `embedDigits2` (0.5 point)

- **Use contextVal= 13 and messageVal=1 as your method arguments**
- One such test case is already given in MethodTester.java to test `embedDigits2`. Your test should follow the same format as the given test case.

A.3 Write a general version of your method named `public static int embedDigitsN(int contextVal, int messageVal, int N)` (0.5 point)

- This method will embed an integer of `N` bits, `messageVal`, in `contextVal`.
- For this method you should assume that `contextVal` will be an 8-bit int between 0 and 255 inclusive, while `messageVal` will be an `N`-bit int between 0 and 2^N-1 inclusive, and that 0 < `N` <= 8.
- This method will also return the result (i.e. the new value after embedding).

A.4 Write a test in MethodTester.java to test *method* `embedDigitsN` when method arguments are `contextVal`= 64, `messageVal`=2 and `N`=5 (0.5 point)

- **Use contextVal= 64, messageVal = 2, and N = 5 as your method arguments**
- One test case is given in MethodTester.java to test `embedDigitsN`. Your test case should follow the same format.

Looking ahead, we're also going to need a way to retrieve the 2 least significant bits from a number in order to recover our secret message...

A.5 Write a method named `public static int getLeastSignificant2(int num)` (0.5 point)

- This method will return the 2 least significant digits in decimal from the input `num`, which will always be an integer between 0 and 255 inclusive.

A.6 Write a test case in MethodTester.java to test method `getLeastSignificant2` when method argument is 63 (0.5 point)

- One test case is given in MethodTester.java to test `getLeastSignificant2`. Your test case should follow the same format.
- **Hint:** Recall the mod operator (%) we've been using over the last few weeks. It'll come in handy again here!

Some sample output:

```
>>> Picture.getLeastSignificant2(18)
2      # 18 is 00010010 in binary and 10 in binary is 2 in decimal

>>> Picture.getLeastSignificant2(0)
0      # 0 is 00000000 in binary and 00 in binary is 0 in decimal

>>> Picture.getLeastSignificant2(251)
3      # 251 is 11111011 in binary and 11 in binary is 3 in decimal
```

Then, you need to generalize the method you wrote in A.5. by writing following method:

A.7 Write a method named `public static int getLeastSignificantN(int num, int N)` (1 point)

- This method will return the `N` least significant digits of parameter `num`.

A.8 Write a test case in MethodTester.java to test `getLeastSignificantN` (1 point)

- **Use num= 28 and N = 4 as your method arguments**
- One test case is given in MethodTester.java to test `getLeastSignificantN`. Your test case should follow the same format.

Part B : Putting it all together (10 points)

IMPORTANT: We have provide some sample images in the starter folder, and you should have copied those images into your folder in section 2.

B.1 Write a method:

`public static Picture hideSecretMessage2Bits(Picture context, Picture message, int x, int y)` **(3 points)**

- This method makes a copy of the `context` image, call it `canvas`, and hides the image `message` inside `canvas` in its least significant 2 bits and then returns this new image.
- At first, you can assume that `message` will always be the same size as `canvas`, but after you get that working, you should modify your code so that it handles images of any sizes.
- **(x, y) is the start point for the hidden image position, which is the left-top corner.**
    - If `canvas` is bigger than `message` in both the horizontal and vertical directions, then `message` should be hidden in the upper left corner of the copy of `canvas` -- **`at start point (x, y)`**.
    - If `message` is bigger than `canvas` in both the horizontal and vertical directions, then it should be clipped so that only the upper left segment of `message` gets hidden in `canvas`. The rest of `message` is ignored.
    - If `message` is wider than `canvas` but `canvas` is taller than `message`, then only the left side of `message` (up to the width of `canvas`) will be hidden inside the top region of `canvas` (up to the height of `message`). The rest of `message` is ignored.
    - If `message` is taller than `canvas` but `canvas` is wider than `message`, then only the top region of `message` (up to the height of `canvas`), will be hidden inside the left side of `canvas` (up to the width of `message`). The rest of `message` is ignored.
- This method should make a copy of the image stored in `context` and *modify and return that copy*. **It should not modify the `context` image which is passed in.** Test your picture thoroughly using the existing code in PSA6. Select different images and see if your method works well for all possible scenarios.

  For example, image A on the next page shows the original `context` image, image B shows the `message` image, and image C shows the returned image from `hideSecreteMessage2Bits` with the `message` image hidden in the copy of original `context` image. **Keep in mind that images A, B, C and D start hiding/recovering at position (0,0), but image E starts hiding/recovering at position (100, 100).**

Image A (context)

Filename: M&M.bmp

Image B (message)

Filename: CookieMonster.bmp

Image C (result)

Filename: M&MWith.bmp

B.2 Write a method:

`public static Picture recoverSecretMessage2Bits(Picture picWithMessage, int x, int y)` **(2 points)**

- This method returns a new picture object, which is the hidden image.
- The method should assume that the message is hidden in the least significant 2 bits of the context image.
- If the message was smaller than the context image, there will be some visual "noise" around the message. If the message was larger than the context, then it will appear clipped.
- Your method won't know the dimension of the hidden image so all you need to do is to go through the entire `picWithMessage` image and visually examine its returned image.
- Write tester codes to test your method in PSA6.java.

  For example, the image on the following page shows the extracted image from image C above after calling the `recoverSecretMessage2Bits` method. As can be seen, image D is slightly different from image B above due to the decrease in the number of colors used in image D.

Image D (extracted message)

Filename: recCookieMonster2bit.bmp

Image E (extracted message)

This image is recovered from (100, 100), which is why it appears clipped.

B.3 Write two more methods:

1. `hideSecretMessageNBits(Picture context, Picture message, int N , int x, int y)`
2. `recoverSecretMessageNBits(Picture context, int N ,int x, int y)`

   (3 points, 1.5 points for each method)

- **Note: The `context` parameter for method `recoverSecretMessageNBits` refers to the image with a picture hidden in it -- i.e. the returned image from `hideSecretMessageNBits`**
- These methods will hide and recover a secret image in the `N` least significant digits in the `context` image
- Test these methods in PSA6.java by taking the parameter N=2.

- **The resulted images should be exactly the same as the ones you got in the above 2 methods** `hideSecretMessage2Bits` **and** `recoverSecretMessage2Bits`**.**

A Note about Testing:

You are provided with 2 files, PSA6.java and MethodTester.java, to help you with the testing. Use PSA6.java specifically to test and show your pictures. Use MethodTester.java specifically to test methods that return int values. This will make debugging easier for you.

TEST YOUR FUNCTIONS CAREFULLY! You can test your hide and recover functions by hiding a message and then recovering it. You should get back an image visually similar to the image you hid (although, of course, your image will look a little degraded because you are using fewer bits to represent color).

You can use `degradeColorsNBits` method to to generalize your steganography methods and examine the extracted image in the end. What pattern do you see with respect to image degradation and N? You can inspect these images visually, or, if you are ambitious, you can write a function which compares the recovered message with the degraded image programmatically.

Small pictures are available in mediasources directory, which can be used to test your implementations.

Part C: Saving your hidden message (2 points)

The methods above do not change the original picture. The method `hideSecretMessage2Bits` gives you a new picture with message hidden in it. You can save that picture using something like this:

```
Picture contextWithMsg = Picture.hideSecretMessage2Bits(context,message, 0, 0);
contextWithMsg.write(System.getProperty("user.home")+"/my_picture_with_hidden_msg.bmp");
```

Similarly, you can save the message returned from the method `recoverSecretMessage2Bits`:

```
Picture msg = Picture.recoverSecretMessage2Bits(picWithMessage, 0, 0);
msg.write(System.getProperty("user.home")+"/my_hidden_msg.bmp");
```

IMPORTANT: You should use bitmap format to write your image, NOT jpg. To do this, just make sure your filename ends with .bmp when you call `write()`. Make sure you have permission to write to your home directory if you are working on your own machine.

After you save the new picture, you should test it by reloading the image with message and extracting the secret image from it.

C.1 Submit at least one image with a secret image hidden in the least significant 2 bits (2 points).

- The name of your image should be *my_picture_with_hidden_msg.bmp*.
- Additionally, you should submit your recovered image and name it *my_hidden_msg.bmp*.
- You **MUST** show the context image, message image, image with hidden message and recovered message image using PSA6.java.

In total, you should submit the following 4 images:

- The original context image: "original.bmp"
- The original message image: "message.bmp"
- The context with the hidden message: "my_picture_with_hidden_message.bmp"
- The recovered message: "my_hidden_message.bmp"

IMPORTANT: As part of this PSA, all hidden images should be appropriate and decent. Students will lose points if any inappropriate picture is used as a secret image.

Program Description (2 points)

Describe what your program does as if it was intended for a 5 year old or your grandmother. Do not assume your reader is a computer science major. The programs you should comment on in this segment include Chromakey.java and Picture.java. **Write this as comments at top of Picture.java file.**

Comments and Style (2 points)

Remember to include file header comments with your and your partner's name, login ID, and date in *all* your files.

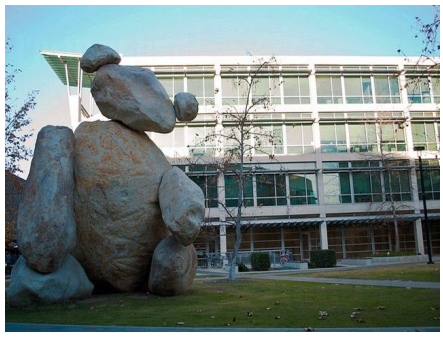
We will be looking for the following additional comment/style points in your code:

- Meaningful variable names (i.e. don't name a pixel "n", "i", or "bob" or "hello". Name them something pertinent to their role, such as "sourcePixel" or "targetPixel")
- Proper code indentation. Every subsequent coding/loop block must be further indented properly. Every single code example in your textbook follows this convention, so see your book for examples
    - In Dr. Java, you can highlight everything and press tab -- it will automatically indent everything for you! Remember for future projects (cse8b, etc.), you won't have Dr. Java as your editor, and will have to do this indenting manually!
- Method header comments at beginning of methods indicating what they do at a high level.
    - Example: **"*//method flips the image upside down*"** or "***//method mirrors the image along the center vertical line***".
    - Include a description of the parameters and return values in the header comments (google javadoc styling if you're not sure).
- **Each code line should be no longer than ~80 characters.**
- **Do not copy-paste formatted text into your code! Apostrophes and certain characters being inside of your code will break the code! We will take points off if your code does not compile because of this!**

Star Point (Optional)

If you want to give your classmates a challenge, create some secret messages that are hidden in different numbers of bits, and leave it to your classmates to discover how many bits they should look in. You can link your picture from the dedicated Piazza post!

In addition, we provide one more stone bear picture (you should have copied the image in section 2). There is a secret message delivered to you from a cute guest, but you need to find the hidden picture to see what that message is (Hint: the least significant 2 bits are important). Find out what the message and cute guest are! :)



Stone bear with a hidden message in it

Filename: StonebearWith.bmp

The star point opportunity for this assignment is to create an additional, innovative way to hide picture/text in a picture and implement it on a picture of your choice. For example, you might hide a text message inside a picture, or hide more than one picture in the same place inside your picture, without losing the color information in either.

Note: Not all attempts will earn the star point. You should do this part only if you are interested.

You will write the star point code in a static method in your Picture class called `starPointEmbed(...)` and `starPointExtract(...)`.

- You should provide the complete logic of how to **embed and extract** secret information.
- You can submit additional pictures named *starpoint_original.bmp, starpoint_message.bmp, starpoint_picture_with_hidden_message.bmp,* and *starpoint_recovered_message.bmp*
- Submit all your logic in a text file called *starpoint.txt* (simply explain how you hid and recovered the message).

These optional files should be in your PSA6 directory while turning in. As usual, you must make it clear who worked on the star point. If both you and your partner chose to implement the star point, but implemented them separately,

you should include *two* star point methods in your single Picture file (`starPoint1` and `starPoint2`) along with two sets of star point images. However, if you both decide to work on the star point, we recommend that you work together!

How to Turn in Your Homework

1. Place your psa6 folder in the home directory of your student account (cs8afxx).
2. Ensure that the psa6 folder contains at least all the following files: **PSA6.java, Picture.java, MethodTester.java, original.bmp, message.bmp, my_picture_with_hidden_message.bmp, my_hidden_message.bmp.** If you attempted the star point, also add the star point files according to the naming rules described in section 6.
3. Run the command: cse8aturnin psa6
4. Follow the prompts.
5. Verify that your homework was turned in with the command: cse8averify psa6

Refer back to psa0 turnin instructions for additional details.