Lab 8: Image manipulation

One thing many people use computers for is editing and sharing photos. In fact, one of the most common ways that "non-programmers" program is to script programs like Photoshop to (for example) produce finished yearbook pages or business cards from a directory full of image files and a text or CSV file of string and numeric information. A few lines of code and a few minutes can replace hours of hand-editing. This part of the lab will introduce a small selection of image manipulation primitives, you can probably think of many more once you get the idea.

You probably have guessed by now that Images (and Strings) are not *really* "atomic", we just presented them that way. If we "open up" the image representation, we can do some interesting things. In particular, Images are really a Complex Itemization [i.e. an Image is either a Circle or a Rectangle or a Bitmap, or `(overlay Image Image)`, or…]. Photos in DrRacket are represented directly as Bitmaps (and any Image can be rendered as a Bitmap). A Bitmap is a structure containing pixels, each pixel represented by a Color structure. A Color is composed of four channels: red, green, blue, and alpha. Channel values range from 0 (totally off) to 255 (totally on/colored). Alpha is transparency, so 0 represents totally transparent and 255 represents totally opaque.

Read though the provided `lab8-extras.rkt` to learn about the various data definitions and functions that are provided to you for working with Bitmaps.

Instructions

Using Sakai, turn in a single file *lab8.rkt* containing all code and documentation for this assignment. Both partner's names must be listed in a comment at the top of the file.

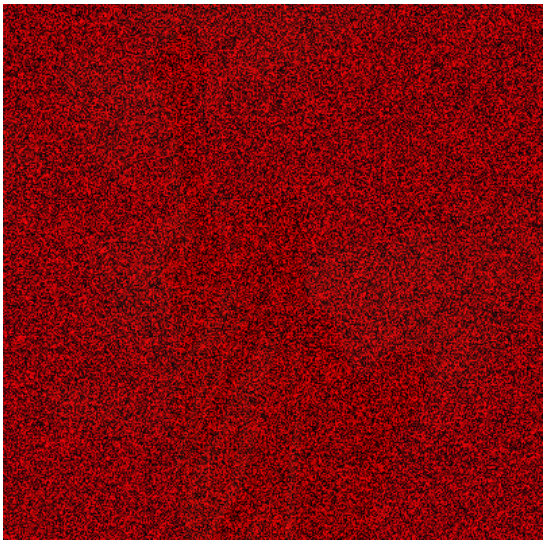
This lab requires lab8-extras.rkt. Download this file from the Sakai Resources area or lab attachments and save it in the same directory in which you are saving lab8.rkt. Then add the following lines to the beginning of your lab8.rkt:

```
(require picturing-programs)
(require "lab8-extras.rkt")
```

Lab 8 is due Oct 26;
Recall that your Project 1 is also due Oct 27

Problem 1: Hidden Image

The COPPER image is a puzzle—it shows something famous, however the image has been distorted. The true image is in the blue and green values, however all the blue and green values have all been divided by 19, so the values are very small. The red values are all just random numbers, noise added on top to obscure things. Undo these distortions to reveal the true image. Develop the function `copper-decode` to undo the distortions and reveal the real image.



Hints: Use `map-image` and `local`!!! Remember that a `ColorNum` is a `NatNum` in the range `[0,255]`, so you may need to `round` values to the nearest integer.

Testing: The builtin function `pixel->image` will create a one-pixel image, with the given pixel color. You can then compare that image to another one-pixel image. So, for example, after changing red channel to 0 and multiplying the green and blue channels by 19, the resulting one pixel image should be

```
(check-expect (copper-decode (pixel->image (make-color 1 1 1)))
              (pixel->image (make-color 0 19 19)))
```

Problem 2: Instant Alpha

Often you want to remove parts of the background of a picture so that you can place it over another part; Mac OS X apps like Keynote and Pages use a tool for this called Instant Alpha (this is also related to "green screen" video special effects). Basically, you select a color, and a range, and then all pixels *near* the chosen color will have their alpha channel set to 0 (making them transparent): anything placed behind the image will now show through.

Distance in color space can be computed in the usual manner (square root of the sum of the squared differences). If Color1 is [r1, g1, b1, a1] and Color2 is [r2, g2, b2, a2] then the distance is:

$$\sqrt{(r2-r1)^2+(g2-g1)^2+(b2-b1)^2}$$

Develop the function `instant-alpha` that accepts as parameters a Color, a Distance, and an Image and produces an Image where the Alpha of all of the pixels whose Color is within the given Distance of the given Color is set to 0.

The extras file includes a function called `sinimage: Number Number -> Image` which will be useful for testing `instant-alpha` by creating a crazy colored background.

For example,

```
(overlay (instant-alpha BLUE 200 YD)
         (sinimage (image-width YD) (image-height YD)))
```

should yield this:

Problem 3: Remove Red-Eye

A similar problem is to remove red-eye caused by a flash. The idea here is to change every pixel *near* the color red with another color (say black). A good value for "near red" is within a distance of 165 to `RED`.

However, we usually only want the eyes to change, most photo editors allow you to select the area of the eyes.

Develop a function `remove-red-eye` that consumes four NatNum arguments corresponding to the bounding box to change (x1 y1 x2 y2) and an image, and produces a new image with any reddish pixel changed to black within that bounding box. For example, on the left is the constant image JEN, and on the right is the result of `(remove-red-eye 100 90 190 130 JEN)`

Problem 4: Pixel as Data

Pixel as Data is an idea by Matthieu Savary and friends at the French design agency User Studio. They have developed many algorithms for creating abstract images using other images as raw data.

In this problem, you should develop some aesthetically pleasing filter `pad` based on the following ideas:

- `image->color-list : Image -> [ListOf Color]`
  converts an image into a list of pixels (colors).
- This list can be manipulated any way you want. Swap channels? Apply a math function? Sort the pixels by some sorting function?
- For further fun, you can preprocess the image using map-image to store (scaled) x or y values in the alpha channel of the color list. You'll need to reverse this at the end to put alpha back to 255/ opaque (unless that's part of your idea…)
- When you are done, the list of pixels can be converted back to a bitmap image using
  `color-list->bitmap : [ListOf Color] Width Height -> Image`
  where Width and Height are the PosInt width and height of the output image.

For example, in the first image I simply sorted the pixels (I'll keep the sort function secret).

In the second, I first encoded the y value in the alpha channel, and then sorted by the same function within each row (so the pixels only change in a single row).

The source image for both is MATISSE [Henri Matisse: Red Madras Headdress (Le Madras Rouge) 1907 Image © 2016 The Barnes Foundation]

Problem 5: Steganography

Steganography is a form of security through obscurity that hides messages in such a way that no one, apart from the sender and receiver, realizes that there's a secret message. A specific type of steganography relies on the fact that humans are generally poor at distinguishing between similar colors. As you've seen, computers tend to represent colors as different amounts of red, green, and blue. By subtly tweaking the values of the color channels we can encode secret messages that are visually undetectable.

a. Write a function `encode` that accepts two images (image and message) of the same size and produces an image. *Each pixel in the produced image should be the same as the corresponding pixel in the given image except that, if the corresponding pixel in the given mask is not WHITE, the red channel should be rounded up to the nearest odd number otherwise the red channel should be rounded down to the nearest even number.* Note that the red channel is chosen because people are better at distinguishing differences in greens and blues.

```
(encode (overlay/align "center" "center"
                       (text "Secret" 24 "black")
                       (rectangle (image-width MATISSE)
                                  (image-height MATISSE)
                                  "solid"
                                  "white"))
        MATISSE)
```

should produce something like:

Notice that, visually, the image looks identical to MATISSE.

b. Write a function `decode` that accepts an image and produces an image. Each pixel in the produced image should be BLACK if the red channel of the corresponding pixel in the given image is odd, otherwise it should be WHITE.

```
(decode (encode (overlay/align "center" "center"
                        (text "Secret" 24 "black")
                        (rectangle (image-width MATISSE)
                                   (image-height MATISSE)
                                   "solid"
                                   "white"))
          MATISSE))
```

should produce:

Note that the shadow is just to show the bounds of the produced image.

Problem 6: The Pesky Tourist

Sometimes you want a picture of a landmark or sculpture, but find it hard to get a picture without anybody in the frame. If those pesky tourists are moving *enough*, you can remove them by using a ***median filter***. Consider the following 9 frames of a sculpture:

If you look closely, you can see that the most of the sculpture is visible in each image. The tourist only blocked part of the sculpture, and since he was in a different position in each photo, he usually blocked a different part of the sculpture each time. *If there was a*

way to determine which pixel colors are part of the sculpture and the background (what you want), and which pixel colors are part of the tourist (what you don't want), you could create a photo without the tourist.

In a median filter, we compare individual pixels from each image. We start out by making the assumption that most of the pixels in each image at a specific coordinate [x,y] are the same color or very close in value. These are the colors we want. Sometimes, at a particular pixel in a particular image, that pixel will be part of the tourist and that pixel will have a very different color from the other pixels at the same coordinate in the other images. These different colored values are the noise we are looking for and trying to remove.

In order to pick the correct color for the pixel at each [x,y] coordinate, we can read the color value for that particular coordinate in all the images at the same time, placing them into a list. We can then sort those values, and pick the middle value, i.e., the median value for that list of numbers. Once we know that median value, we can write it out to our filtered image. Of course we have to remember that each pixel really contains three values: one for red, one for green, and one for blue. That means for each pixel, we have to find three median values.

If we do the above process for each pixel, and if we do it for the red, green, and blue value in each pixel, then the final result will create a good photo without that pesky tourist.

Problem a: Design a function `median` that consumes a non-empty list of numbers (parametric data definition is `[NEListOf Number]`) and produces the median value. As my middle schoolers remind me, to find the median you sort the list, and then if the length is odd, take the middle value. If the length is even, then take the average of the middle TWO values. *Hint: use* `local` *to avoid recomputing things, and remember that* `(list-ref list i)` *gives back the i'th element of the list, where 0 is the first element's index.*

```
(check-expect (median '(1))       1)
(check-expect (median '(1 3 2))   2)
(check-expect (median '(1 3 4 2)) 2.5) ;2.5=(2+3)/2
```

Problem b: Design a function `make-median-color` that consumes a list of `Color` and produces a new `Color` were the red value is the median red value, the green value is the median green value, and the blue value is the median blue value (alpha should remain fully opaque, so just use the three-argument version of `make-color`)

Now, before we design `median-image`, we need to consider, again, efficiency. The function `get-pixel-color` is (barely) optimized for a single image, and will be extremely slow across a list of 9 images or more. The way to think about this problem is that we can think of each image as a *list of pixels*, and only work on (find the median of) one pixel, of every list, at the same time. This is another general abstract looping construct, `map-list`.

For example, consider adding up the columns in a spreadsheet. Each row is a list of Number, and then the spreadsheet is a list of rows:

```
;; SHEET is a [ListOf [ListOf Number]]
(define SHEET (list (list 10 20 30)
                    (list 15 10 11)
                    (list  5 10  9)))
```

If we want to add the columns, we need to map a function over all the first elements, then all the second elements, then all the third elements, etc. (all the rows have the same length). Here we want the totals `(list 30 40 50)` from adding each column. If we have a function `sum` that adds a list of numbers:

```
;; sum: [ListOf Number] -> Number
(define (sum alon) (foldr + 0 alon))
```

Then we just need to call sum on the list of first elements, the list of second elements, etc. and cons the results together into a returned list. The parametric signature of map-list is

```
;; map-list: ([NEListOf X] --> Y) [NEListOf [ListOf X]] --> [ListOf Y]
```

In our spreadsheet example, Type X is Number, and the final result Type Y is also a Number, so `sum` and SHEET (which is non-empty) work as arguments of the correct Types:

```
;; map-list: ([NEListOf Number] --> Number) [NEListOf [ListOf Number]]
;;           --> [ListOf Number]
(check-expect (map-list sum SHEET) (list 30 40 50))
```

`map-list` is already written for you and provided by `lab8-extras.rkt`. Now we can finish this final problem.

Problem c: Design a function `median-image` that consumes a list of images (parametrically, `[ListOf Image]`) that are all the exact same size (width and height) in pixels, and produces a single `Image` where every pixel is the median of the corresponding pixel in the original list of Images.

Hints:

1. The predefined function `image->color-list` converts a single image into a list of pixels (`[ListOf Color]`)
   Signature: `image->color-list: Image -> [ListOf Color]`
2. The predefined function `color-list->bitmap` converts a list of pixels/colors into an image of the given PosInt width and PosInt height
   Signature: `color-list->bitmap: [ListOf Color] PosInt PosInt -> Image`
3. The predefined functions `image-width` and `image-height` find the width and height, respectively, of a given Image.
4. You already wrote `make-median-color`, that collapses a list of colors/pixels into one representative median color/pixel
   Signature: `make-median-color: [ListOf Color] -> Color`
5. The predefined function `map-list` calls some function on a list of every first element, every second element, …every nth element…, and collects the result of those function calls into a list. What are Type X and Type Y here?
6. Follow the Design Recipe. The body of this function can be written using just local and function composition of these existing parts.

Extra Credit

In order for median-image to be really useful you would want it to process a set of files, and write out a final file. To do this, you would need to have some info about the file names, and use that to grab all the images out of files, call the function in problem 6, and write out the result to a new file.

Design a batch program `main` function that consumes a base filename (a String), an extension (“png”, “jpg”, or “gif” are supported by 2htdp/images) and a number of files, and produces a single file that is the result of calling median-image on all the files.

- Assume that the files are sequentially numbered, so if base is “Tower” and extension is “jpg” and number of files is 3, then the filenames are exactly
  - “Tower1.jpg”
  - “Tower2.jpg”
  - “Tower3.jpg”
- Write the result out to the file “Tower-median.png” (this is the only supported output format)
- Look up these `2htdp/image` functions to read and write Images:
  - `bitmap/file : String -> Image`
  - `save-image : Image String -> Boolean`