Creative Project 2: Mondrian Art

Specification

(Much of this assignment inspired by Ben Stephenson's 2018 Nifty Assignment.)

Background

In recent years, algorithmic art has become more commonplace, ranging from very simple, largely randomly generated pieces to sophisticated works created by artificial intelligence programs such as DALL-E. The artwork created by these systems can be quite impressive, but the process is not without criticism — in particular, many worry that the increase in computer-generated art, especially for commercial purposes (such as logos, advertisements, cover art, etc.) may have a significant impact on the livelihood of professional artists.

In this assignment, we will attempt to programmatically create images that evoke the style of 20th-century Dutch abstract artist Piet Mondrian, specifically pieces such as Tableau I and Composition A. While our program will not be particularly intelligent, it will follow certain rules that attempt to emulate some of the techniques Mondrian used.

> **i** **NOTE:** In this assignment, we are in no way intending to claim we can *recreate*, or even *approximate*, the work of artists like Mondrian, nor are we claiming that the images we will create comparable to Mondrian's works. Rather, the images we will create are simply meant to *evoke* the style of a subset of Mondrian's work by focusing on certain elements and techniques.

Learning Objectives

By completing this assignment, students will demonstrate their ability to:

- Define a solution to a given problem using a recursive approach
- Write functionally correct recursive methods
- Produce clear and effective documentation to improve comprehension and maintainability of a class
- Write a class that is readable and maintainable, and that conforms to provided guidelines for style, and implementation

Assignment Details

Program Overview:

You will implement a program to generate random artwork in the style of Mondrian. Your program will:

- Start with a blank canvas of specified dimensions
- Recursively divide the canvas into smaller regions until they reach a minimum size (specified below)
- Color the regions randomly to mimic Mondrian's style.

The specifics of this algorithm should be as follows:

- If the region being considered is at least one-fourth the height of the full canvas **and** at least one-fourth the width of the full canvas, split it into four smaller regions by choosing one vertical and one horizontal dividing line at random.
- If the region being considered is at least one-fourth the height of the full canvas, split it into two smaller regions by choosing a horizontal dividing line at random.
- If the region being considered is at least one-fourth the width of the full canvas, split it into two smaller regions by choosing a vertical dividing line at random.
- If the region being considered is smaller than one-fourth the height of the full canvas **and** smaller than one-fourth the width of the full canvas, do not split the region.

Any time a region is split, the dividing line(s) should be chosen at random to be within the bounds of the region. You should ensure that dividing lines are chosen such that each resulting subregion is at least 10 pixels by 10 pixels. In other words, dividing lines should not be chosen too close to the boundaries of a region.

> ✅ **HINT:** If you want to pick an integer between `a` inclusive and `a + b` exclusive then you can call `rand.nextInt(b) + a`.

Once a region is below a certain size, it should be filled in with a color chosen randomly from red, yellow, cyan, and white. When filling a region, leave a one-pixel border around the edge uncolored -- this will give the appearance of black lines separating the colored regions.

> ℹ️ **Accessibility Note**
> We acknowledge that the assignment may not be accessible to those who are visually impaired or colorblind. If this is a concern, please reach out for additional support by emailing the instructors or making a private post on the Ed board and we will make alternative arrangements for you.
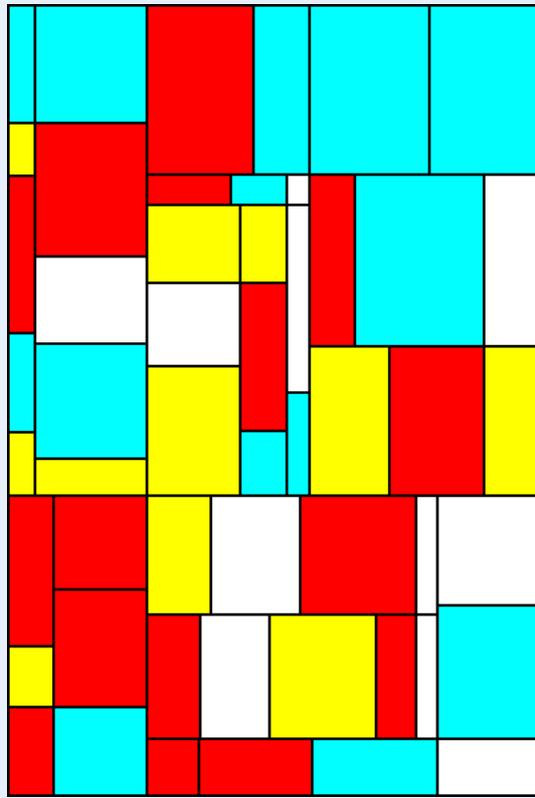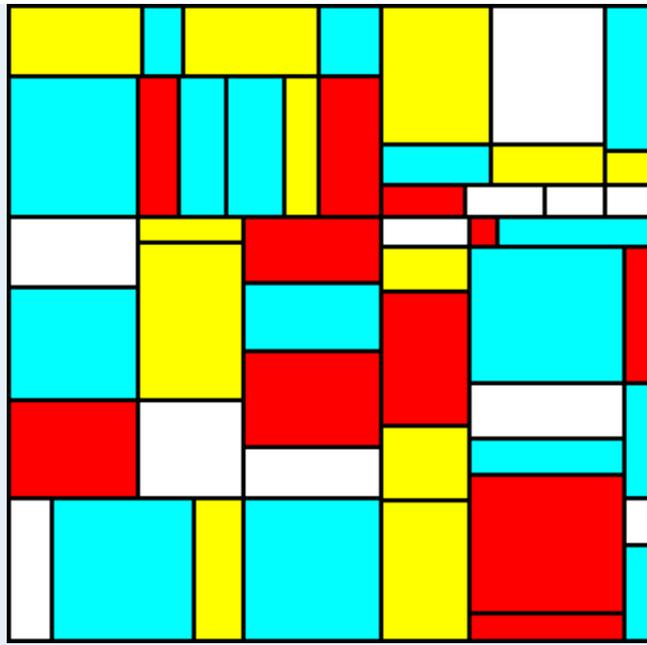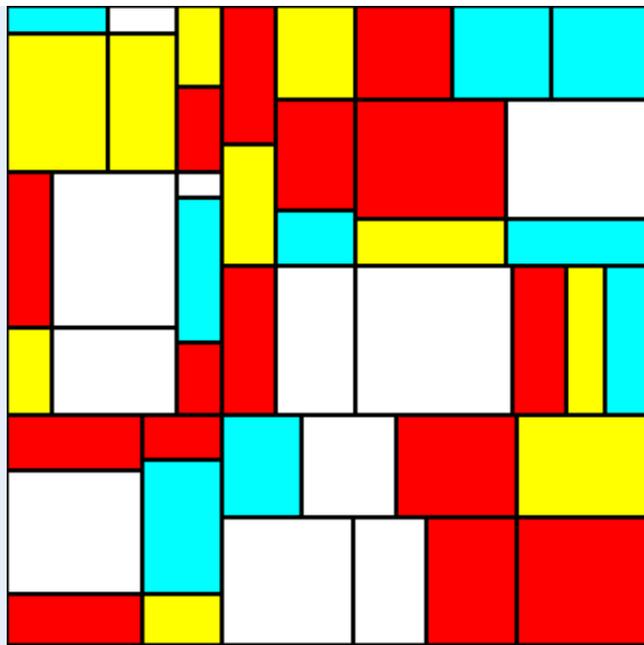
Example Images

Here are a few examples of the images your program might generate:

> ℹ️ It may not be clear from these images, but there is also a black border around the entire canvas. You should not limit the border to interior edges.

▼ Expand

Extensions

To earn a grade of E on this assignment, you must implement a creative extension that meets the following requirements:

- The core of your implementation is recursive.
- Your output is non-deterministic (random). That is, the output is different each time the program is run.
- Your design incorporates at least two different colors.
- Your output is clearly different from your basic Mondrian.
    - As a rule of thumb, your grader should be able to tell whether they have run your basic or complex Mondrian based only on what the output looks like.

You may implement any extension you like that meets these requirements. If you need ideas, here are a few options you could consider:

Color Related to Location

▼ Expand

Instead of determining the color of a region completely randomly, allow a specific color to be *more likely, but not guaranteed,* to occur in a region of the overall canvas.

For example, you might make regions closer to the upper left more likely to be red and regions closer to the lower right more likely to be blue. This would make most, but not all, of the boxes in the upper left red, and most boxes in the lower right blue. You could also make regions closer to the center of the image more likely to be darker and regions closer to the edges more likely to be lighter. (You should still color each full region a single color.)

ℹ️ For this option, you may want to use the `Color` constructors rather than simply relying on the constants. See the "Working with Images" slide for more information.

Random Number of Splits

If a region is large enough to be split in a dimension, instead of always splitting into two smaller regions on that dimension, split the region into a randomly chosen number of smaller regions, with a maximum of 4 splits on each dimension. If the region is large enough to be split on both dimensions, you should choose the number of splits on each dimension separately

Random Fill Shape

If a region is not large enough to be split, instead of always filling the entire region as a rectangle, randomly choose among several shapes to fill in that region. You may choose the specific shapes that will be possible, but there should be a least 2 such shapes (of which a rectangle may be one).

Implementation Requirements

Required Methods

You must implement the following two methods in the provided `Mondrian` class in your assignment:

```
public void paintBasicMondrian(Color[][] pixels)
```

- Fill `pixels` with `Color` objects (using the `java.awt.Color` class) according to the basic algorithm specified above.
- If `pixels` is null, throw an `IllegalArgumentException`
- If the length or width of `pixels` is less than 300, throw an `IllegalArgumentException`

```
public void paintComplexMondrian(Color[][] pixels)
```

- Fill `pixels` with `Color` objects (using the `java.awt.Color` class) based on your chosen extension.
- If `pixels` is null, throw an `IllegalArgumentException`
- If the length or width of `pixels` is less than 300, throw an `IllegalArgumentException`

Your methods should work properly when called with the provided client program in `Client.java`. You are welcome to modify the client for your own testing if you would like, but your methods must function correctly with the provided code.

`java.awt.Color`

You can directly access `Color` objects to represent the colors needed to create the assignment through the `static` fields of the `Color` class. For example, a `Color` object that represents red can be accessed by writing `Color.RED`, or the `Color` object that represents yellow can be accessed through `Color.YELLOW`.

```
Color shapeColor = Color.RED;
```

The four colors that you will use for the basic part of this assignment (`paintBasicMondiran`) can be accessed by writing:

`Color.RED`, `Color.YELLOW`, `Color.CYAN`, `Color.WHITE`

You may choose to use additional colors for your extension (`paintComplexMondrian`).

Recursion

To earn a grade higher than N on this assignment, **your algorithms must be implemented recursively.** You will want to utilize the *public-private pair* technique discussed in class. You are free to create any helper methods you like, but the core of your algorithm (specifically, the dividing of regions) must be recursive.

Remember that when we say "your algorithms must be recursive" we do not mean "loops are forbidden". We simply mean that the main part of the algorithm must be recursive. You can still use loops where appropriate, as long as they contribute toward your main recursive algorithm.

Since recursion with public-private pairs often leads to many parameters that need to be tracked and updated, we encourage you to factor logic regarding randomly dividing regions into a private helper method or private helper class.

Testing

For this assignment, you are not required to write any tests, JUnit or otherwise.

Assignment Requirements

For this assignment, you should follow the Code Quality guide when writing your code to ensure it is readable and maintainable. In particular, you should focus on the following requirements:

- Avoid making objects you don't need. Look for opportunities to construct objects and reuse

them elsewhere. This can be tricky to spot, but in particular, you should review your code statements that instantiate `new` objects or call other methods that do so.

- Look out for including additional base or recursive cases when writing recursive code. While multiple calls may be necessary, you should avoid having more cases than you need. Try to see if there are any redundant checks that can be combined!
- Make sure that all parameters within a method are used and necessary.
- You should comment your code following the Commenting Guide. You should write comments with basic info (a header comment at the top of your file), a class comment for your `Mondrian` class, and a comment for every method.
  - Make sure to avoid including *implementation details* in your comments. In particular, for your object class, a *client* should be able to understand how to use your object effectively by only reading your class and method comments, but your comments should maintain *abstraction* by avoiding implementation details.
  - Continuing with the previous point, keep in mind that the client should **not** be aware of what implementation strategy your class/methods utilize.
- All methods present in your class that are not listed in the specification must be private.

Recommended Approach

Breaking It Down

Similar to P0, we've crafted a series of related subproblems to what the spec is asking you to do that are simpler and meant to help you build your way up to your final program. This step-by-step approach is designed to make the learning process more manageable and less daunting. It also should help make the process of working with images a bit easier, as this is likely territory that hasn't been explored deeply in prior courses.

Our Recommendation

We strongly recommend using the sequential subproblem slides. It's a step-by-step journey that breaks down the complexity into digestible parts that will hopefully make it a smoother learning experience!

Specification

The next slide is the **Specification** detailing the requirements of the main assignment. The subproblems may seem unrelated to the final goal, but we promise that they'll help! Each slide will provide details on how to use your previously working subproblem to help simplify the next level of complexity. We will build up toward the final **Mondrian Art** slide, where you will see all your hard work come together to complete the full assignment!

Working with Images Tutorial

In the assignment detailed on the following slide, we will use recursion and image manipulation techniques to create some interesting computer-generated art. The following tutorial explains how to work with images in Java. The information in this tutorial will be helpful for completing this assignment.

Representing Images

> ℹ️ The following section contains links to further resources if you are interested in learning more, but only the information written on this slide is needed for the assignment.

Before you can manipulate digital images, you need to learn about how they are represented. Don't worry though, there are only 2 components to learn about - pixels and color!

Pixels

Pixels are the building blocks of digital images. Each pixel can be thought of as a square of a color that we choose. If we arrange these pixels in a grid, or an array, and choose the color of each one in a meaningful way, we can produce a digital image! By making these pixels small enough, you will see a continuous image instead of the individual pixels that compose it.

Color

In order to accurately and easily represent the wide array of colors that we see every day, a simple way to construct a color using a few base parameters is needed.

RGB

The RGB color model is an additive system in which the red, green, and blue colors of light are added together in specific intensities, reproducing a huge number of colors. This model is based on the theory of trichromatic color vision that our eyes use to interpret light, and has been used since the late 1800s in early color photography.

How RGB works

By assigning a numeric value to each RGB component of a color, we can easily store color values on a computer. Most digital colors can be described through a combination of 3 integer values, ranging from 0 - 255 inclusive, that specify the intensities of the RGB components of said color. For example, a shade of purple can be described with the values `R=106, G=13, B=173`, and a shade of gold can be described with the values `R=255, G=215, B=0`. These values make sense when you think about

them; purple is the result of mixing red and blue, and so you would expect its red and blue RGB components to be higher than its green. Since yellow is the result of mixing red and green, gold's blue RGB component is 0, and its red and green are large numbers.

Expand to see colors:

▼ Expand

106, 13, 173

255, 217, 0

Giving each pixel of an image its own RGB color allows us to represent incredibly complex images. For each pixel on the image you want to display, all you need to store are the intensities of the RGB components of the color you want, and voila! You have an image!

Modern displays and monitors can directly control the level of the RGB colors displayed within each pixel, and so our images can now be loaded and displayed directly on most modern computers!

If you would like to play around with RGB colors to see what different combinations of values create, feel free to use this RGB color picker, or discover a tool for yourself online.

Image Manipulation in Java

Now that you know the fundamentals of how digital images are made, you can learn how to make and manipulate them in Java! The first thing to learn is how Java represents colors. To do this a library aptly named `Color` is used.

Color

`Color` is a class provided in the `java.awt` package that combines code used for creating, representing, and manipulating colors in Java.

The `Color` class is able to create new colors from their RGB values alone. To do this the following syntax is used:

```java
Color myColor = new Color(255, 217, 0);
```

The three `int` values that we pass to the `new Color()` call are respectively the R, G, and B components of the color we are trying to create, in this case, a shade of gold. The values that we pass in must be in the range of 0 - 255 inclusive or an `IllegalArgumentException` will be thrown.

> ℹ️ Because `Color` is from the `java.awt` package, when we use it we must remember to add the following import statement to the top of our program:
> ```java
> import java.awt.*;
> ```

The `Color` object also allows us to get the individual R, G, and B components of the color it is representing. Given a `Color` object `myColor` created above, you can do this through the following commands:

- `int redComponent = myColor.getRed();`
  - Returns the red component of the `Color` as an `int` value from 0 - 255 inclusive.
- `int greenComponent = myColor.getGreen();`
  - Returns the green component of the `Color` as an `int` value from 0 - 255 inclusive.
- `int blueComponent = myColor.getBlue();`
  - Returns the blue component of the `Color` as an `int` value from 0 - 255 inclusive.

Using the above information, you can now create any given RGB color in Java. You can also modify any existing one by retrieving its components and making a new color based on those values.

The Color class also comes with several constants for common colors, so that we can use those colors without having to find their RGB values. For example, a `Color` object that represents red can be accessed by writing `Color.RED`, or the `Color` object that represents yellow can be accessed through `Color.YELLOW`. Expand below for a full list of the available constants:

> ▼ Expand
>
> `Color.BLACK`, `Color.BLUE`, `Color.CYAN`, `Color.DARK_GRAY`, `Color.GRAY`, `Color.GREEN`, `Color.LIGHT_GRAY`, `Color.MAGENTA`, `Color.ORANGE`, `Color.PINK`, `Color.RED`, `Color.WHITE`, `Color.YELLOW`

Picture

The last fundamental of digital images we need to learn about in Java is pixels and their arrangement. We will be taking advantage of an external library to help you easily make and manipulate digital images. This library, named `Picture.java`, gives us access to a new object, the `Picture`, that allows you to represent an image using a 2D array of `Color` information. Before we break down exactly how this 2D array works, let's look at the functionality of the `Picture` class. It is important to note that for the rest of this assignment, `Picture` refers to the Java object, and the image refers to the digital image that we are creating.

Creating a `Picture`

There are 3 useful constructors implemented by `Picture`. You can use them to either create a `Picture` from an existing digital image, from an existing `Picture` object, or you can create a blank one with a specified size. Here is the syntax for these means of creation:

- `Picture myPicture = new Picture("image.jpg");`
  - This syntax lets us pass in the name of an existing image as a `String`, in this case `"image.jpg"`. The resulting `Picture` is an exact RGB representation of that image. The image that you pass in must exist in the same directory level as the program you are writing, similar to the requirements for reading from a file. The following image formats are supported by this library (.png, .jpg, and .gif).
- `Picture myPicture = new Picture(500, 300);`
  - This syntax lets us pass in two `ints` representing the desired width and height of the image you want to create. The resulting `Picture` is a blank image of the specified dimensions. Both width and height must be greater than 0, and an `IllegalArgumentException` will be thrown if they are not.
- `Picture myPicture = new Picture(otherPicture);`
  - This syntax lets us pass another `Picture` object to make a copy of the original picture with its own state of pixels.

In this assignment, you will primarily be using the second method of creating an image, but feel free to experiment with creating images from scratch on your own.

> **i** Because `Picture` is from an external library, we must make sure that the `Picture.java` file is present in the same directory as the program we are working on. This is done for you on Ed, but make sure to download `Picture.java` from Ed and place it in your working directory if you are programming in your own environment.

Picture Methods

There are three more methods associated with `Pictures` that you will need to work with digital images. Given the `Picture` object `myPicture` from above, you would use these methods in the following ways:

- `myPicture.save(String filename);`

- - This will take the `Color` data for each pixel from `myPicture` and save it to the given file name.
  - `Color[][] pixels = myPicture.getPixels();`
    - Returns a 2D array of `Color` objects. Each element within the array represents the color of the corresponding pixel in `myPicture`. This is used to retrieve the RGB data from an image so that you can edit it.
  - `myPicture.setPixels(Color[][] newPixels);`
    - Replaces the `Color` data of the Picture with the data from the passed-in array. Each element in the `newPixels` array represents the new color of the corresponding pixel in `myPicture`.

Using the above information you can now create images and display them on our screen using Java! With the `getPixels()` and `setPixels()` methods you can also retrieve, edit, and replace the Color data of any image you want! Before showing examples of how to do this, let's talk about the 2D arrays that you will be using to make your images.

2D `Color` Array

In order to understand 2D arrays, we often visualize them as an indexed grid. It turns out that this is pretty much exactly how images are interpreted! Digital images are essentially just a grid (or 2D array) of pixels, so it is possible to directly represent an image as an array of pixels. A pixel is essentially just the color data for a small segment of an image, and so all you need to store for each pixel is its color. Since colors are represented using the `Color` library in Java, our image can be directly represented with a 2D array of `Colors`! Written in Java, this is a `Color[][]`.

In less complex terms; an image can be represented by a grid, where each point in the grid has a specific color value. When these colors are displayed in grid order, they appear as an image. A 2D array of `Colors` can be used to maintain this grid and achieve this effect. For example, we show an example image as a grid of pixels, each one storing its own color RGB value. Now this picture is "low resolution" since it is 4 pixels by 5 pixels. A real image might have thousands, if not millions of pixels in it; with them so small it's very hard to see them with the human eye.

In our representation, the top-left pixel of an image lives at the indices `(0, 0)`. So if our `Color[][]` was called `pixels` then `pixels[0][0]` refers to this top-left pixel. As the first index increases, that moves down the image and as the second index increases, that moves to the right. In the picture above, the following locations have the following indices:

- Top-left: `pixels[0][0]`
- Top-right: `pixels[0][4]`
- Bottom-left: `pixels[3][0]`
- Bottom-right: `pixels[3][4]`

Putting it all together

Using what you've just learned about working with `Colors` and `Pictures` in Java, and with your previous knowledge of 2D arrays, you can now create and manipulate your own digital images!

Here is an example program that takes an image, manipulates its RGB values, and then shows the new image to the user. We will break down how this program works afterward:

```java
import java.util.*;
import java.awt.*;

public class Main {
    public static void main(String[] args) {
```

```java
        // Step 1: Load image: Note we have to use a slightly
        // different file name in a reading slide. In your code
        // you can just say "suzzalo.jpg" or whatever the name
        // of that photo is in your directory
        Picture pic = new Picture("/course/suzzallo.jpg");
        pic.save("originalSuzzalo.jpg");

        Color[][] pixels = pic.getPixels();

        // Step 2: Change image
        // See below for code
        increaseRed(pixels);

        // Step 3: Set pixels and display Image
        pic.setPixels(pixels);
        pic.save("angrySuzallo.jpg");
    }

    // This method increases the red RGB component of each pixel
    // within the passed in image. Takes the Picture to edit as
    // a parameter.
    public static void increaseRed(Color[][] pixels) {
        for (int i = 0; i < pixels.length; i++) {
            for (int j = 0; j < pixels[i].length; j++) {
                Color originalColor = pixels[i][j];
                int red = originalColor.getRed();
                int green = originalColor.getGreen();
                int blue = originalColor.getBlue();

                Color newColor = new Color(Math.min(red + 100, 255), green, blue);
                pixels[i][j] = newColor;
            }
        }
    }
}
```

The `main` method of this program does 3 things.

1. Creates a new `Picture` by loading an image (`"suzzallo.jpg"`) and retrieves the pixel values for the image.
2. Calls a method, `increaseRed()` , on the `Color[][]` which represents the pixels of the image to boost the red component of the image's RGB values.
3. Sets the pixels of the `Picture` to be the new pixels we have edited then calls `.show()` on the `Picture` after the changes to show the image to the user.

You could also describe the actions of `main` by saying it loads, then changes, then displays an image. Your program will mimic this 3 step structure for its main method, although step 2 will be more complicated.

Changing the Color of a `Picture`

Let's dig deeper into the `increaseRed()` method, starting with the method header:

```
public static void increaseRed(Color[][] pixels) {
```

Notice that we take a `Color[][]` as a parameter, and that our return type is `void`. Remember that with arrays, changes made to them as a parameter effect them in their original scope too. Because of this, we do not have to return a resultant image from our manipulation methods.

The first thing this method does is use the `Color[][] pixels` to retrieve the needed pixel information. This data is given to us as a 2D array of `Colors`. The nested `for` loop technique that we covered in class is used to iterate through the 2D array in order to process the individual pixels.

At each pixel, we perform the following steps:

1. Retrieve the `Color` for the pixel directly from the 2D array. You can do this using typical array syntax: `Color originalColor = pixels[i][j];`
2. Retrieve the individual RGB components from the original color, using `originalColor.getRed()`, `originalColor.getGreen()`, and `originalColor.getBlue()`.
3. Create a new `Color` object for the pixel using the RGB values you took from the original color as a basis for your new color. Any manipulations are applied in this step.

   Notice the call to `Math.min()` for the red component of this new `Color`. Since we are increasing the red component of each pixel by 100, there could easily be a case where the resultant value is greater than 255. Since 255 is the largest value for any RGB component, we cannot increase red beyond this. In order to cap the increase to 255, we can call `Math.min(red + 100, 255);`.

   `Math.min()` works by taking 2 values and returning the smaller one, so this call will always return a value less than or equal to 255.
4. Replace the old Color in the 2D array with the new one that you just created. You can do this using typical array syntax: `pixels[i][j] = newColor;`

After both `for` loops have finished running, our image has been fully processed and the control flow of our program can go back to `main`. Notice that the `main` method of the program contains the calls of `pic.getPixels()` and `pic.setPixels(pixels)`. By extracting the pixel data at the start of our program and applying the changes onto the `Picture` at the end, we can hold off the expensive computation of updating our `Picture` until all changes to the pixels have been made. It might not seem more efficient now, when we are only applying a single filter to our image, but in the next parts of this assignment you will be applying many filters onto your image, and taking advantage of this technique will increase the speed of your program significantly.

You could describe the total actions of our program more simply like this:

1. Get the 2D `Color` array (the pixels) from the `Picture`
2. For each filter, at each pixel:

1. Get the individual RGB values from the `Color`
2. Make a new `Color` based on those values but with the desired manipulations made
3. Replace the old `Color` in the 2D array with the new one
3. Once you have gone through every pixel with each filter, replace the old 2D `Color` array in the `Picture` with the new one you have just edited.

The next slides describe image manipulation methods and algorithms that you will be implementing. Remember to refer back to the information above while writing these. You can roughly copy the structure of the `increaseRed()` method to do most tasks in this assignment.

Spec Walkthrough Video

An error occurred.

Unable to execute JavaScript.

[NOT GRADED] Fill Region

Download Starter Code:

📄 C2_Fill_Region.zip

The first step of working our way up to a complete Mondrian solution is meant to get you used to working with images, as they haven't been strongly introduced before. In this slide, you'll implement fill, a method which fills in a provided rectangular region with the color white within the image. Below is the full specification:

```
fill(Color[][] pixels, int x1, int x2, int y1, int y2)
```

- Fills in `pixels` within the region defined by `x1`, `x2`, `y1`, and `y2` where (x1, y1) represents the upper-left corner *inclusive*, and (x2, y2) represents the lower-right corner *exclusive*. Your solution should leave a single pixel as a border around the edges of the region (*you can optionally ignore this border requirement*).
- We recommend you approach this method iteratively

Note that you'll also have to set up the `main` method to properly call this method and display the results. If you're ever confused about how to do something with the provided `Picture` class, check out the `Working with Images Tutorial` slide above!

⚠️ **WARNING:** Note that `pixels` is a 2d array. This means that the first index represents the current row and the second index represents the current column. If you think about these as coordinate plane values (x, y) they'll be reversed when dealing with the `pixels` array: `pixels[y][x]` gives the pixel at point (x, y).

More information on the [Working with Images Tutorial] slide!

ℹ️ **NOTE:** We have 3 different tests depending on how you want to think about borders in this problem. **It's impossible to write a solution that passes all 3**. You only need to pass at least one before moving on to the next subproblem.

[NOT GRADED] Divide Canvas

***Download Starter Code:***

📄 C2_Divide_Canvas.zip

ℹ **NOTE:** This is not intended to be a direct translation to the final Mondrian submission. This is just a problem we recommend you do to help practicing recursion with images (like `weave` for `synchronize`)

The next step of working our way up to a complete Mondrian solution is meant to help you in using *recursion* with images. In this slide, you'll implement `divideCanvas`, a method that divides the given picture into 4 equal-sized regions `n` times. Below is the full specification:

```
divideCanvas(Color[][] pixels, int n)
```

- Divides the given `pixels` (assumed to be all black) into 4 equally sized regions `n` times, filling in each of the split regions with the color white while leaving a 1 or 2 pixel border along the edges of each of the resulting regions.
- You may assume that `n` is greater than or equal to 0.

For example, calling `divideCanvas(pixels, 0)` would not create any divisions and color the canvas as one big region with a border:
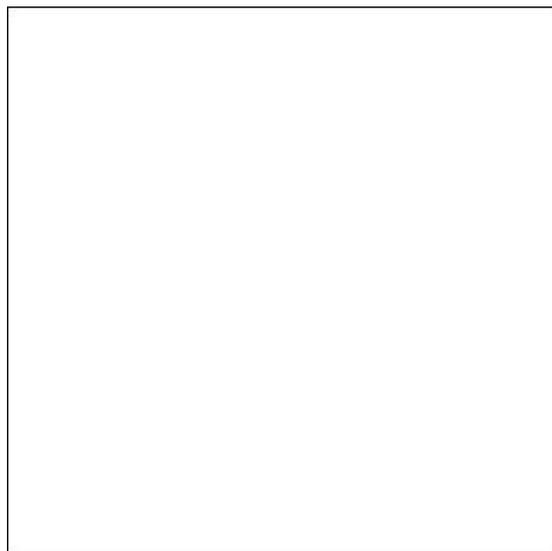
Calling `divideCanvas(pixels, 1)` would split the canvas once, into four equally-sized regions like below:
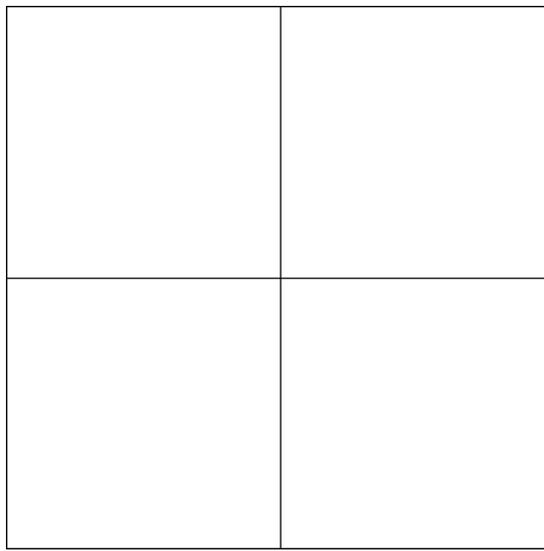
Calling `divideCanvas(pixels, 2)` would result in a picture where the four larger regions are each split into four subregions:
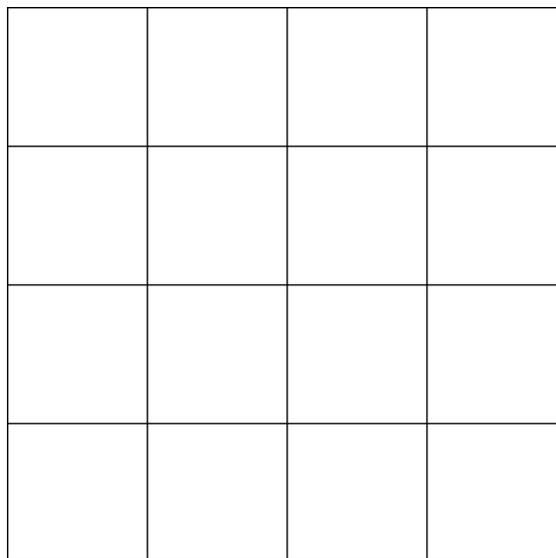
> ✅ **HINT:** Here you'll want to keep track of regions somehow within your solution. Think about how you represented a region in your `fill` solution and how you can do something similar here!

Mondrian Art

Download Starter Code:

📄 C2_Mondrian.zip

If you used the previous development slides, we'd encourage you to turn to `divideCanvas` for inspiration (particularly regarding dividing regions). However, note that this problem is substantially different from `divideCanvas` meaning you **should not copy-paste your working `divideCanvas` solution into this slide.** Rather, you should start from scratch applying what you previously learned.

Required Methods

You must implement the following two methods in the provided `Mondrian` class in your assignment:

```
public void paintBasicMondrian(Color[][] pixels)
```

- Fill `pixels` with `Color` objects (using the `java.awt.Color` class) according to the basic algorithm specified in the Specification slide.
- If `pixels` is null, throw an `IllegalArgumentException`
- If the length or width of `pixels` is less than 300, throw an `IllegalArgumentException`

```
public void paintComplexMondrian(Color[][] pixels)
```

- Fill `pixels` with `Color` objects (using the `java.awt.Color` class) based on your chosen extension.
- If `pixels` is null, throw an `IllegalArgumentException`
- If the length or width of `pixels` is less than 300, throw an `IllegalArgumentException`

Your methods should work properly when called with the provided client program in `Client.java`. You are welcome to modify the client for your own testing if you would like, but your methods must function correctly with the provided code.

⚠ **WARNING:** The tests provided are **not** considered to be exhaustive. You should still run your program to ensure that the outputted image meets the expected requirements.

Reflection

For this week's reflection, we would like everyone to watch and engage with the following video.

Step 1: Watch Why Artists are Fed Up with AI Art. (12m 55s)

An error occurred.

Unable to execute JavaScript.

Question 1

Step 2: Write your own comment below, responding to one of the following prompts in its entirety:

Prompt 1: How do you think the increased prevalence and sophistication of AI-generated artwork affect the livelihoods of traditional artists? Consider established artists, who created work that AI algorithms train on, as well as aspiring artists, who have to compete with computer-generated options. Do you think AI art will influence the appreciation of human-created artworks, and why?

Prompt 2: How might AI-generated art influence cultural narratives and the preservation of artistic heritage? In what ways could the proliferation of AI art impact public perceptions of creativity and originality? How can societies ensure that the integration of AI in art benefits a diverse range of communities and does not exacerbate existing inequalities?

Prompt 3: Do you think any of the issues discussed in the video, or any other issues of AI- vs. human-created art, are different for different art forms (e.g. music, fiction writing, poetry, dance, visual art)? Why or why not? What types of creative expression do you think are most or least susceptible to disruption from AI?

Prompt 4: How do you think algorithmic art (like what you created in this assignment / what was discussed in the video) compares to human-created art? List at least one advantage and one limitation of algorithmic art compared to human-created art?

Please label which prompt you are answering. For full credit, all questions within a prompt must be clearly answered. In particular, we suggest using the ACE (answer, cite, and explain) format. For a meaningful response, it may be helpful to pose some counterexamples, connect in terms of your own experience, add additional support from course materials, and be as specific as possible in your own reasoning

Question 2

Step 3: The following questions will ask that you practice **metacognition** to reflect on the topics covered on this assignment and your experience completing it. For each question, focus on your plan and/or process for working through the assignment along with the CS concepts. Think about things like how you organized your working time, what sorts of things tended to go wrong, and how you dealt with those errors or mistakes.

Describe how you went about testing your implementation. What specific situations and/or test cases did you consider? Why were those cases important?

Question 3

Do you think the algorithm you used to create your images would have been easier, harder, or neither to implement using iteration instead of recursion? Why?

Question 4

What skills did you learn and/or practice with working on this assignment?

Question 5

What did you struggle with most on this assignment?

Question 6

What questions do you still have about the concepts and skills you used in this assignment?

Question 7

About how long (in hours) did you spend on this assignment? (Feel free to estimate, but try to be close.)

Question 8

Was any part of the specification or requirements unclear? If so, which part(s), how was it unclear, and how could it have been made more clear?

Question 9

[OPTIONAL] Do you have any other feedback, questions, or comments about this assignment?

(Note that we may not be able to respond to questions here, so please post on the message board if you would like a response!)

Final Submission

Fill out the box below and click "Submit" in the upper-right corner of the window to submit your work.

Question

I attest that the work I am about to submit is my own and was completed according to the course Academic Honesty and Collaboration policy. If I collaborated with any other students or utilized any outside resources, they are allowed and have been properly cited. If I have any concerns about this policy, I will reach out to the course staff to discuss *before* submitting.

(Type "yes" as your response.)