CS 145- Lab 5

Overview:

This lab will give your group practice using recursion. Lecture videos covering recursion are posted on Canvas if you missed the recursion lectures in class. This lab is adapted from Ben Stephenson's (University of Calgary) "Random Mondrian Art" assignment, accessed via Stanford's nifty website. This is my first time giving this lab so if something seems way too difficult/generally off, or the writeup in the spec doesn't make sense, come talk to me.

If you have been a lesser contributing member of your lab group recently, I encourage you to step up and take the lead on this lab. Especially Part II! Recursion is a completely new topic and this lab doesn't build off of objects, data structures, and other topics we have covered so far in 145. You can succeed with this lab by understanding recursion even if some earlier course concepts are fuzzy. And Part II requires more creativity than extensive coding knowledge.

Lab Groups:

You will continue to work with your assigned lab groups. Each team member is expected to contribute to the successful completion of this lab. One member of your lab group will upload the group's code to codePost and generate a link to add the other group members as partners. Only team members who use this link and add themselves to the CodePost submission will receive credit for the lab.

Background:

Piet Mondrian (March 7, 1872 – February 1, 1944) was a Dutch painter who created numerous famous paintings in the early half of the previous century that consisted of a white background, prominent black horizontal and vertical lines, and regions colored with red, yellow and blue. Three examples are shown below:

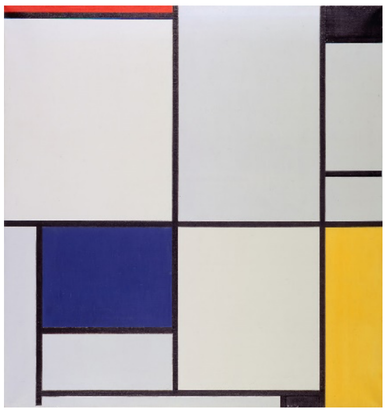
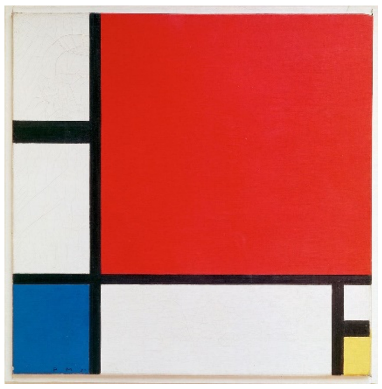
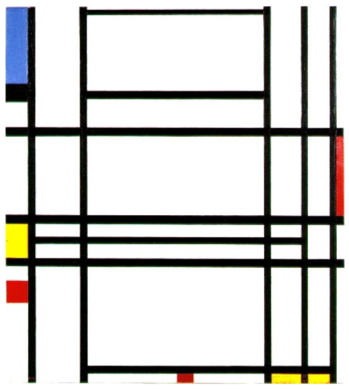


Tableau I, 1921


Composition II in Red, Blue, and Yellow, 1930

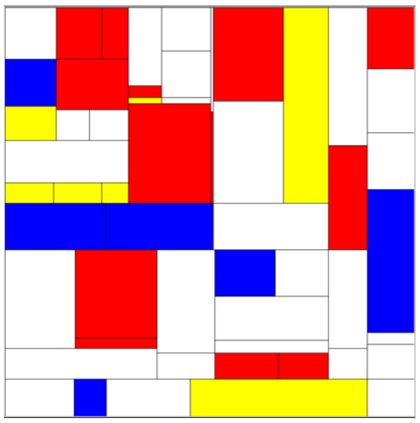

Composition No. 10, 1939-1942

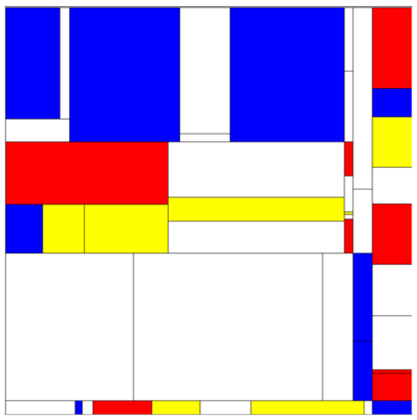
Lab Task- Part I:

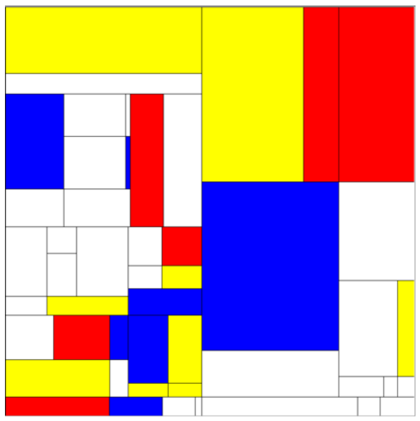
Your task is to write a Java program that uses recursion to generate pseudo-random "art" in a Mondrian style. Your program must be named MondrianArt.java and it will display a DrawingPanel (details below) with filled and non-filled rectangles. The following general strategy will be used to generate art in a Mondrian style:

```
If the region is wider than half the initial canvas size and the region is taller than half the initial canvas height:
    Use recursion to split the region into 4 smaller regions (a vertical split and a horizontal split) with both split locations chosen randomly.
Else if the region is wider than half the initial canvas size:
    Use recursion to split the region into 2 smaller regions using a vertical line with the split location chosen randomly.
Else if the region is taller than half the initial canvas size:
    Use recursion to split the region into 2 smaller regions using a horizontal line with the split location chosen randomly.
Else if the region is big enough to split both horizontally and vertically (details on what that means below pics), and both a horizontal and vertical split are randomly selected:
    Use recursion to split the region into 4 smaller regions (a vertical split and a horizontal split) with both split locations chosen randomly.
Else if the region is wide enough to split horizontally, and a horizontal split is randomly selected:
    Use recursion to split the region into 2 smaller regions using a vertical line with the split location chosen randomly.
Else if the region is tall enough to split vertically, and a vertical split is randomly selected:
    Use recursion to split the region into 2 smaller regions using a horizontal line with the split location chosen randomly.
Else:
    Fill the current region (randomly, either white or colored, and if colored, with a random determination of red, blue, or yellow).
```

A couple of images I generated using this algorithm are shown below.







Use the following strategy when randomly deciding whether or not to split a region:

> Generate a random integer between 100 and the width of the region * 1.5.
> If the random integer is less than the width of the region then split the region.

While this strategy works, you might find yourself asking: why is the random number between 100 and the width of the region * 1.5? By using 100 as the lower bound for the random number, we ensure that we never split a region that is less than 100 pixels wide (or tall when splitting in the other direction), and as such, are sure that the region is big enough to split (for my arbitrary definition of big enough). Selecting a random value that could be up to 1.5x the width of the region, but then only performing a split when the random value is less than the width of the region, provides a random chance that a larger region will not be split into smaller regions.

Use the following strategy when splitting a region, either because it is so big that it will always get split, or because it was randomly selected to be split:

> Choose the split point, randomly, somewhere between 33% and 67% across the region (or down the region if splitting in the other direction). Choose two random split points when splitting both horizontally and vertically.
> Split the region into two smaller regions, one on the left and one on the right (or one on top and one on the bottom), or four smaller regions if splitting both horizontally and vertically.
> Use recursion to fill/further split each new region.

Save a copy of your program that implements the above algorithm as MondrianArt.java. You will receive up to a 75% on the lab if you stop here. Part II builds off the above algorithm but be sure you have a copy of your work that stops here to turn in. Make a copy of your file with a new name to build off of for Part II below.

Implementation Strategy:

- Start small! (Re)familiarize yourself with using DrawingPanel (details below) to draw basic shapes.
- After that, start trying to write your recursive method. Start with the base case and one simple recursion case. Get your program splitting the grid into squares recursively with no randomness.
- Start adding in one piece of randomness at a time. Maybe start with randomly filling or not filling the squares.
- Add in another piece, maybe randomly filling the squares with color.
- Add randomness to where you split your box, creating rectangles rather than squares.

- Slowly add in one more piece at a time until you have successfully implemented the entire algorithm above.

Using DrawingPanel:

DrawingPanel was created by UW professors for simple graphical output. You likely used it for the Café Wall Lab in CS& 141. DrawingPanel can be downloaded from the Building Java Programs website at http://www.buildingjavaprograms.com/drawingpanel/DrawingPanel.java

The funky thing to know for DrawingPanel is that the (0, 0) x and y coordinates are at the top left of the panel rather than the bottom left like typical cartesian planes. An example of using DrawingPanel can be found in the "example basic usage" section of DrawingPanel's javadocs. Note that to use DrawingPanel successfully you will utilize Java's Graphics object which will require you to import java.awt.*; Details on fillRect(), drawRect(), setColor() and other potentially useful drawing methods can be found in Graphics' javadocs. The three methods just mentioned should be all you need graphics-wise for Part I of the lab.

VSC sometimes struggles to properly compile new files added to a folder when the folder is already open. If you run into compilation issues after downloading DrawingPanel.java, try File -> Open Folder and re-open the folder containing your code and DrawingPanel.java.

Part II:

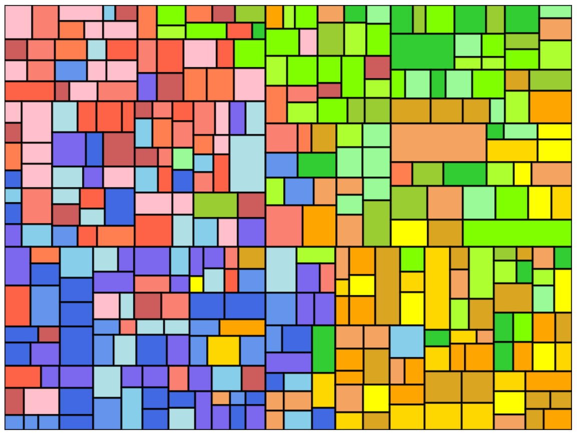
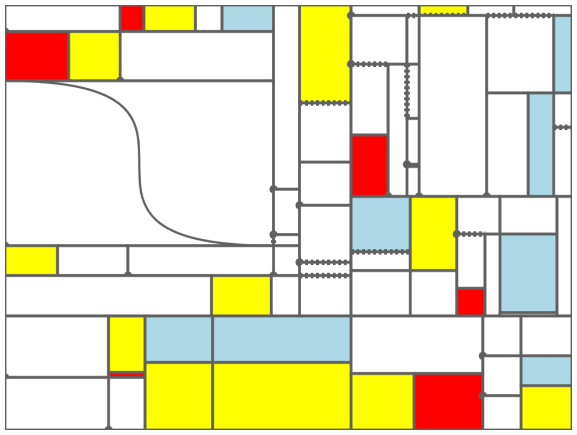
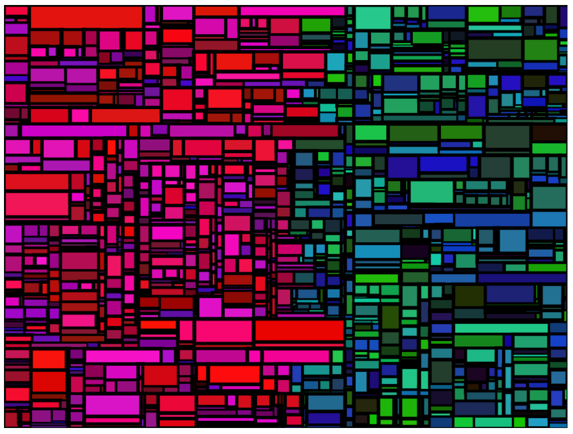
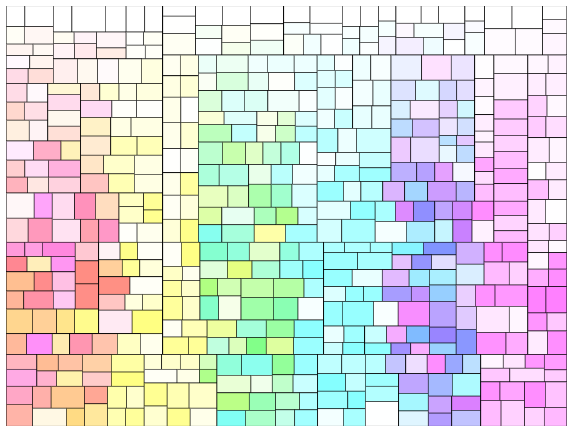
Once you have the provided algorithm in Part I working, you are to make a copy (you will be turning in two files) of your program and adjust/expand/adapt this algorithm to generate art in your own specific style. Your art must still be at least vaguely Mondrian in style (meaning that it largely consists of horizontal and vertical lines and colored regions, at least the majority of which are rectangular). Ideas for customizing your work that you might want to consider include:

- Using lines of variable width
- Using a broader color palette than red, yellow and blue for the filled regions
- Changing the distribution of random numbers used when selecting the sizes of regions, colors (or anything else random). For example:
  - Instead of using numbers that are an even random distribution, you could add two random numbers together and divide by two to form a distribution that favours numbers in the middle of the random space.
  - Instead of allowing all possible values, reduce the space to numbers that are evenly divisible by 10 (or 20 or some other number) so that the random lines have more regular spacing to them.
- Using a patterned fill for some regions instead of only using solid fills

- Occasionally splitting a region into something other than rectangles
- Occasionally split a region into 3 smaller regions instead of 2 or 4

To get full credit for this part of the lab you must make a minimum of **two significant modifications** to your Part I code to generate images that look significantly different than Part I. Changing a single color or two does not count as a significant modification.

For inspiration, here are a couple of images that University of Calgary students & faculty generated with different variations of the Part I algorithm above. In some of these images, the color used to fill the region is influenced (but not fully determined) by its location:

Implementation Details:

You must properly define at least two class constants for use throughout your program. One (or two, your panel can be a rectangle rather than a square if desired) is the size of your DrawingPanel, and another is the minimum threshold to stop splitting your region again (100 as described above).

Your Part I code may not use any loops or data structures. For Part II, you can use extras such as these if they add to your own art style but the basis of your program must still run recursively. You can add helper methods if desired but the foundation of your program should be main calling into a method that recursively calls itself.

Use Java's Random class as needed and feel free to do any casting necessary to convert from doubles to ints or vice versa. You need to use parameters, not class/instance variables (declared above main) to make your recursive code work properly. The only class/instance variables you should have are your class constants.

Use good programming style as taught throughout the quarter. Review the CodePost comments from prior labs and assignments and make sure you don't make any of the same mistakes you've made before again. Points (or more points if some were already deducted) will be deducted increasingly throughout the quarter if the same mistakes are repeated again and again. Examples of stylistic deficiencies and other undesirable behaviour that might result in a deduction of points include (but are not limited to):

- Repeated code
- Magic numbers
- Missing or low quality comments
- Poor method/parameter/variable names
- Crashing
- Generating useless output (such as rectangles that are outside of the canvas)

Sample Solution Length:

- It should require less than 100 significant lines of code (excluding comments, blank lines, and lines with only curly braces on them) to implement Part I

Extra Credit:

- You can earn up to 2 points of extra credit for implementing the Part I algorithm iteratively (this means using loops). Technically, anything that can be implemented iteratively can be implemented recursively and vice versa. Some problems lend themselves to one implementation style or the other much better (such as recursion with Mondrian style art) but both are possible. Fair warning, the iterative code for this algorithm will likely be long and ugly.
- You can earn up to 2 points of extra credit for creating another *recursive* program that generates and displays completely different fractal art. Google fractal trees/n-trees, Sierpinski triangles, Mandelbrot sets, etc. for ideas. You may use loops/data structures as needed but the foundation of your program must be recursion. If there is evidence of plagiarizing someone else's code or using ChatGPT to write this extra credit for

you, you risk receiving a 0 on the entire lab (and possible reporting to WCC for academic dishonesty).

What To Turn In:

- MondrianArt.java from Part I containing the provided algorithm which generates an image in the Mondrian art style. Failure to submit a version of the program that implements the provided algorithm will cause me to assume that any differences occurred because you weren't able to implement the algorithm correctly rather than due to artistic choice.
- A second file with a name of your choosing from Part II that generates your own Mondrian style art image with at least two significant modifications from your Part I code to create images that look interestingly different than Part I.
- Optionally, screenshots of any favorite images generated by your program. Since there is randomness at many different levels, your program will likely generate a significantly different image each time you run it. If you have a favorite image generated by your program that you want to show off, you can turn it in to me.
- An extra file (or two) if you choose to do either extra credit option.

Rubric:

Part I	15
Part II	5
Total	**20**