Zachary Hightower Project 1 Report

The overall method is one that tries to prioritize simplicity and the Java coding fundamentals I've been taught so far. It makes use of 2D arrays to contain the image's properties, keep track of checked pixels, and simulate the area over which the grow-region occurs.

To illustrate all this, I'll go through the code in detail, showing references as I go, and try to explain why I did everything I did in the program.

~

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.Arrays;
import java.util.ArrayList;
```

~

The imports here are pretty standard for reading over any given file. The only two I would say are of note being Arrays and ArrayList. Arrays are here because I learned how to use them during Java II and III for going over matrices like the one described by the pgm we're dealing with. The ArrayList is here because I need something easy to use to hold a big list that I want to add a bunch of stuff to and then, eventually, pump it into another Array so that I can sort it without needing to go through writing a sort or importing anything else.

~

```
//Global variable declaration
private static int width;
private static int height;
private static int targetIntensity;
private static int[][] image2DPixelized;
private static boolean[][] checkedPixels;
private static int[][] TwoD_List;
private static final int Magic_Num = 2;
```

~

There isn't much to see here. I just tried to pick variable names that are essentially full explanations of what they're going to be used for. I also have the Magic Number as a constant here for easy use, though I only ended up using it in one place in the program itself.

~

```
public static void main(String[] CommandLineArgument) {
   //Main method that contains our sub methods. The program is split out into these sub methods
   // so that it's easier to read and easier to debug.
   if (validateCommandLineArguments(CommandLineArgument)) {
      //I'm including the prints for the programmer's name and the file name here, because it's a
little
      // clearer than having it be part of the print for the values, I think.
      System.out.println("Zachary Hightower");
      System.out.println(CommandLineArgument[0]);
      targetIntensity = Integer.parseInt(CommandLineArgument[1]);
      processImage(CommandLineArgument[0]);
   } else {
      //This helps to provide a good error message for any inputs that end
      // up in the wrong form. It gives what failure it is, and an example of how to fix it.
      System.out.println("User input failure, argument must be of the form \n java WIP_growregion
```

```
<image file name> <pixel intensity value 0-255>");
    }
}
```

~

I think everything here is explained already by the comments within the code block reference. There isn't that much to the main method. I'm just using it like it was used in the programs in Java II, a place to stick all the necessary calls to other methods that accomplish what I need to get done.

~

```
private static boolean validateCommandLineArguments(String[] args) {
    //A quick method to validate the CL argument
    return args.length == 2;
}
```

~

I was trying to split things up into as many separate methods as possible. I didn't do it as much as the program went one, because I didn't want to as much as the program went on because it's a little bit of a bother at times.

~

```
private static void processImage(String filename) {
    //This is our method for processing the image fully.
    //It's mostly a place where the reading, region growth, and error handling
    //can be consolidated in a clean, presentable method.
    try {
        readImageFile(filename);
        findRegionsAndPrintResults();
    } catch (FileNotFoundException e) {
        System.out.println("ERROR READING IMAGE FILE: " + e.getMessage());
    }
}
```

~

This portion is a lot like the main method, in that it's mostly here for calls to other methods. It calls over to the method for reading the actual image file, and the method that is, essentially, the end of the program. The only other thing it might do is throw an error if the image file isn't what it's looking for, or it hits any other error that causes it to jump over to the catch block.

~

```
private static void readImageFile(String filename) throws FileNotFoundException {
    //This method is where we read in the image file and make it into a usable 2D array
    //Though before we actually do that, we run the method for gathering the
    //dimensions of the image.
    File file = new File(filename);
    Scanner fileScanner = new Scanner(file);

    initializeImageProperties(fileScanner);
    //This is where we populate the 2D array with pixels from the image
    //We also begin an array of checkedPixels
    //These two function like the board of a battleship game and the pegs that you
    //stick into the board to indicate hits.
    for (int rowIndex = 0; rowIndex < height; ++rowIndex) {
        for (int columnIndex = 0; columnIndex < width; ++columnIndex) {
            image2DPixelized[rowIndex][columnIndex] = fileScanner.nextInt();
```

```
            checkedPixels[rowIndex][columnIndex] = false;
        }
    }

    fileScanner.close();
}
```

~

This block is largely explained by the comments. I picked out using the file object and scanner because that's the thing we did in java II whenever we needed to handle information from files.

The rest of it is initializing things that we're going to use later. The height, width, pixels matrix for the image, and making sure that all the checkedPixels matrix cells read as false, before we actually begin to use them. Then we make sure to close the fileScanner, because otherwise it will stay open and cause problems.

~

```
private static void initializeImageProperties(Scanner fileScanner) {
    //This is the method that gives the variables for the image properties in the
    // program the actual values they need. It also initializes our two arrays used in the
    // prior method and another List used later in the program, which I had to name
    // TwoDList, because Java got angry at me having a 2 in front of it
    // and told me to change that. The last arraylist is essentially the area of the image
    //in the columns, and the Magic Number of the image in the rows.
    String[] WandH_String = fileScanner.nextLine().split(" ");
    width = Integer.parseInt(WandH_String[1]);
    height = Integer.parseInt(WandH_String[2]);
    image2DPixelized = new int[height][width];
    checkedPixels = new boolean[height][width];
    TwoD_List = new int[Magic_Num][height * width + 1];
}
```

~

The comments here say most of what's necessary. The only thing that it might not is that the stringify and regex are being used because of the way that the structure of the image is written about in the pdf. Whitespace separator means we need to use the " " to skip over that and grab what we actually need and place it into usable cells within our string array. We also instantiate here the TwoD_List that I tend to think of as the area coverage, since it uses what is essentially the formula for area in the columns portion of the 2D array matrix.

~

```
private static void findRegionsAndPrintResults() {
    //This method is the one that calls our search function, and prints the results
    //We used nested loops again to go through row by row, column by column in the
    //2D matrix we have.
    //Once we've grown the region, we call the print function.
    ArrayList<Integer> recordedIntensities = new ArrayList<>();
    int numberOfRecordedIntensities = 0;

    for (int rowIndex2 = 0; rowIndex2 < height; ++rowIndex2) {
        for (int columnIndex2 = 0; columnIndex2 < width; ++columnIndex2) {
            if (image2DPixelized[rowIndex2][columnIndex2] == targetIntensity && !
checkedPixels[rowIndex2][columnIndex2]) {
                int regionSize = growConnectedRegion(rowIndex2, columnIndex2);
                recordedIntensities.add(regionSize+1);
                ++numberOfRecordedIntensities;
            }
```

```
      }
   }

   printResults(recordedIntensities, numberOfRecordedIntensities);
}
```

~

I think it should read once we've grown all the regions, we call the print function, but it's too late for me to care too much about such a small tweak.

Anyway, this portion is where we use our ArrayList and look through the image, grow the spots we need to grow, and record everything about that. If the spot in the matrix is what we want, meaning it has the targetIntensity and is not already checked, we grow the connected up region and pop that into our storage int for regionSize, then we add the size of that to our ArrayList for region sizes, adding a plus one to avoid some of the off by one errors I saw in results when running this. Then we increment our amount of recorded intensities to keep track of how many regions we've actually grown in the entire image.

After we create our things to print, we print them.

~

```
private static int growConnectedRegion(int startRow, int startColumn) {
   //This is the method that actually grows the regions based on the targetIntensity
   //picked by the command line argument.
   int regionSize = targetIntensity;

   checkedPixels[startRow][startColumn] = true;
   TwoD_List[0][0] = startRow;
   TwoD_List[1][0] = startColumn;
   int listIndex = 0;

   while (listIndex >= 0) {
      int listValue0 = TwoD_List[0][listIndex];
      int listValue1 = TwoD_List[1][listIndex--];

      //Growing out of the region's pixels around the one we've picked.
      for (int rowIndex3 = -1; rowIndex3 < 2; ++rowIndex3) {
         for (int columnIndex3 = -1; columnIndex3 < 2; ++columnIndex3) {
            int neighborRow = listValue0 + rowIndex3;
            int neighborColumn = listValue1 + columnIndex3;
            if (isValidPixel(neighborRow, neighborColumn) && !checkedPixels[neighborRow]
[neighborColumn]) {
               checkedPixels[neighborRow][neighborColumn] = true;
               if (image2DPixelized[neighborRow][neighborColumn] == targetIntensity) {
                  ++regionSize;
                  TwoD_List[0][++listIndex] = neighborRow;
                  TwoD_List[1][listIndex] = neighborColumn;
               }
            }
         }
      }
   }

   return regionSize;
}
```

~

The method here uses a breadth first search method, and treats the image like a graph. It's based somewhat on the implementation I found online when searching for things to give me an idea of how to tackle this problem.

Reference here:
https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

I wanted to use something like this based on previous problems I've seen during graph node coloring and vertex traversal problems from my 301 course in Discrete mathematics.

~

```
private static boolean isValidPixel(int row, int column) {
      return row >= 0 && row < height && column >= 0 && column < width;
   }

   private static void printResults(ArrayList<Integer> recordedIntensities, int
numberOfRecordedIntensities) {
      //This is the print method. Though it does slightly more than printing by first sticking all
      // the recordedIntensities from our ArrayList into another array so we can use the in-built
      // sort on it and have a neatly arranged result as indicated by the results shown in the
      // pdf. Then it prints everything out in the same format as shown in the pdf.
      Integer[] intensitiesArraySortable = recordedIntensities.toArray(new Integer[0]);
      Arrays.sort(intensitiesArraySortable);
      System.out.print(numberOfRecordedIntensities);

      for (Integer integer : intensitiesArraySortable) {
         System.out.print(", " + integer);
      }
      System.out.println(" ");
   }
}
```

~

This is the last of the blocks. I combined these two in one reference because the first is rather short. All it does is validate the fact that something is, in fact, a pixel, according to what we've found based on height and width.

The final part takes the values that we've found and sorts them for ease of reading. I didn't want to import anything else, so it just throws it in an array and calls the sort associated with them in Java's library associated with it.

It then prints everything out using standard print methods, making sure to add a new line at the end, so that the next command line doesn't print on the same line and make everything look all jumbled together.