Algorithms, continued: another recursive search
Spring, 2021
An image processing example

• Groups of adjacent pixels of the same color define a region (or blob) of an image
• [https://robotacademy.net.au/lesson/multiple-image-regions/](https://robotacademy.net.au/lesson/multiple-image-regions/)
• Let’s try out the connectivity analysis needed here: finding adjacent pixels to a given pixel and checking if they belong to the same region—it’s a recursive search.
Weiss sets up the simplest case: 2 colors (page 346)

• First challenge: Given a grid point in a region, find the number of pixels in the region
Weiss sets up the simplest case: 2 colors (page 346)

- But the problem is not yet fully defined—is that diagonal pair one region or two?
- The video (and Weiss) define two pixels as adjacent if they share an box-edge, so no.
Pixel coordinates

- From mathworks.com: MATLAB docs: The origin is in the upper left corner of the image. The first coordinate is the vertical one, the row number (like matrices in math)

For example, the data for the pixel in the fifth row, second column is stored in the matrix element (5,2).
Data Structure for image

- So the natural data structure for the given pixels is a 2-D array of color, here just 0 or 1 (or could be boolean).
- int color[i][j] = color of pixel in row i, column j
- This should be an instance variable of a class, private of course

```java
public class Image {
    private int color[][]; // pixel array
```
Image Processing: Image Object

- We need to initialize this array based on a provided array—it’s provided when we create the Image object, so it should be an argument to the constructor, like this:

  ```java
  public class Image {
      private int color[][][]; // pixel array
      public Image(int[][][] pixels) {
          color = pixels;
      }
  }...
  ```

- Sample call, for 2x2 image with color values 0 and 1:

  ```java
  Image myImage = new Image({{0,1},{1,0}});
  ```
Sample Problem: given this Image and (x,y) for one pixel, find the number of pixels in its region.

This is an action on an Image, so should be a method of Image, say

int regionSize(int x, int y)

Implementation: We can use greedy!

Just visit the neighbors of the pixel, see if they are the same color, add them to the growing region.
int regionSize(int x, int y)

• Visit the neighbors of the pixel, see if they are the same color, add them to the growing region.
• But watch out, might go into infinite recursion if step out, then back in again...
• So need to make sure it’s a *new* element of the growing set
• Need Set<Pixel> for this...
Class Pixel

- A pixel (pixel position) is just an (int x, int y) pair
- We only need it inside Image, so it can be a private inner class (or even a static one)
- Need equals and hashCode so we can use HashSet<Pixel>
private class Pixel {
    int x, y;

    public Pixel(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object o) {
        if (o == null) return false;
        if (o.getClass() != getClass()) return false;
        Pixel other = (Pixel) o;
        return (other.x == x) && (other.y == y);
    }

    public int hashCode() {
        return x ^ y;
    } // one way: use XOR
}
public int regionSize(int x, int y) {
    Set<Pixel> foundSet = new HashSet<Pixel>();
    theColor = color[x][y];

    But wait, we can’t call regionSize recursively and pass this set down, because it only has x and y as parameters, but we need a recursive search, what to do?

    Answer: cook up a recursion helper, as we saw with mergeSort. It should have a Set<Pixel> argument, plus know theColor and where it’s working…
The recursion helper

private void regionSizeHelper(Set<Pixel> foundSet, int theColor, int x, int y)

- Check if \((x,y)\) is inside the image*, return if not
- If the pixel at \((x,y)\) has color \texttt{theColor}:
  - Create a Pixel for it
  - Check if that pixel is already in the \texttt{foundSet}**
    - If not, add it, call the helper on its four adjacent pixels

*note we can find the array sizing from the color object: \texttt{nrows} = \texttt{color.length}, \texttt{ncols} = \texttt{color[0].length}

**note need of \texttt{Pixel.equals} based on \texttt{x} and \texttt{y} here.
Finally, Call the helper

```java
public int regionSize(int x, int y) {
    Set<Pixel> foundSet = new HashSet<Pixel>();
    int theColor = color[x][y];
    regionSizeHelper(foundSet, theColor, x, y);
    return foundSet.size();
}

Full code
```
Execution

- First call to helper: puts that pixel in the foundSet, calls on 4 neighbors
Execution

Of four recursive calls, only one finds it’s inside, makes 4 calls (some of the original 4 are yet to be executed, will be called later)
Execution

Of four doubly-recursive calls, only one finds it’s inside, makes 4 calls (again, some of these are not yet executed)
Execution

Of four triply-recursive calls, only one finds it’s inside, it makes 4 calls, none of which find anything, so return, return, return, back to top level, finishing up the delayed calls (which don’t find anything)
In general...

- We see that the recursive search only dives down when it finds something, so the depth is limited to the number of pixels in the region. Also the number of leaves in the call tree is limited to $4 \times \#\text{pixels}$. Since the number of interior nodes of the call tree is less than the number of leaves, the number of nodes in the call tree is $O(N)$ where $N = \#\text{pixels}$ in the region.

- At each such call node it does $O(1)$ work, so its performance is $O(N)$, where $N$ is the number of pixels in the region.

- In a sense, this is using DP: we are remembering the progress we have already made, as we go. But let’s classify this as greedy.