Assigned: Thursday 11 March 2010
Due: Friday 19 March 2010
Topics: Recursion
Collaboration: This homework assignment is to be completed individually.
Submission: Follow the instructions for submitting programs via P-Web and handing in a printed copy. Be sure to generate unit tests and a statement justifying their sufficiency.

Background

One common operation for scientists (and image analysts) is to count the number of blobs in an image. For instance, say we have a bacterial colony in a petri dish like the left image below.

![Bacteria in a petri dish](image1.png)

On the right is a processed image that shows each of the things in the petri dish as white blobs. Scientists often want to do things like count the number of blobs, measure how big they are, or get a detailed list of these blobs. Doing these sorts of things manually (or asking graduate students to do them) can be very tedious. Fortunately, you have the tools at your disposal to efficiently handle this problem! The image on the right is binary—consisting of only zeros and ones—and this is easily represented as a two-dimensional boolean array.

For the purposes of this assignment, we will say that two squares in a grid are connected if they share an edge. Thus, each square has at most four neighbors, the squares directly above and below, and to the right and left. Diagonally adjacent squares are not considered connected neighbors. (Note that boundary squares have even fewer neighbors.) If two occupied squares (the individual white pixels of the image above) of the grid are connected, we will say they form a group.

Preparation

1. Copy files for this lab so that they are easily accessible:

   ```bash
   cp ~/weinman/courses/CSC207/code/blobs/* your-dir
   ```

2. You will have copied a barebones class `Grid.java`, three example grid files, and a `GridFactory.java` class for constructing `Grid` instances from the files. Consider the simple grid in `grid.txt`. There is one group of three occupied squares, and one group of an individual occupied square.

3. Open `Grid.java` and look it over to see what it contains.

4. Compile the skeleton class `Grid.java`. 
5. The static GridFactory.read(InputStream in) method is a factory that takes an input stream and creates a Grid object for you. Verify that this works by running the main method already present in the Grid class, which reads a grid from standard input, and outputs the result of grid.toString() to the terminal.

```java
java Grid < grid.txt
```

6. Note the member variable boolean squares[][] . The value at squares[row][col] indicates whether the grid square is “occupied” (i.e., contains a white pixel in the image above). Another view of such a grid may be found in Figure 7.30 of your textbook (p. 346).

**Assignment**

**Measuring Group Size**

Your first task will be to measure the size of a group, given one square of the group. Add a Grid object method

```java
public int groupSize(int row, int col)
```
to the Grid class that does this. If the square at the location is unoccupied, the method should return zero.

To implement groupSize, you will need to recursively visit the neighbors of a square. However, you will not want to visit (and thus count) a square more than once. Thus, you should write a helper method accepting an additional parameter that tracks squares have been visited

```java
private int groupSize(int row, int col, boolean[][] visited)
```
with a post condition that visited[row][col] is set to true.

**Counting Groups**

Add another Grid object method

```java
public int getNumGroups()
```
that counts the number of connected groups in the grid. The method would return 2 for the file grid.txt.

**Hint:** Iterate over the square locations. When you find an occupied grid location, you’ve located a group. Re-use the helper method above as bookkeeping to track whether a particular square has already been counted as part of a group or is part of a new group.

**Listing Groups**

Add another Grid object method

```java
public Set<Set<String>> getGroups()
```
that creates a Set of all the groups present in the grid. Each element in the returned set should be another set of Strings that represent the location of a grid square, i.e. “(1,3)” for the square at row 1 and column 3. For instance, on the file bookgrid.txt, (which corresponds to Figure 7.30), the command getGroups().toString() would produce something like (using the built-in toString method of the Java collections)

```java
[[[3,4]], [[7,5], (7,4)], [(1,9), (0,9)], [(3,7), (5,8), (5,7), (4,7)],
 [(1,4), (1,3)], [(4,3)]]
```
where the square locations in a group are given by (row, col) pairs in a String.

There are several ways to accomplish this. Like the previous problem, you can re-use your helper procedure for groupSize.

**Hint:** Once you find an unaccounted for square, mark its members with your helper and then add them to a set of locations for the group.