9.8 PROJECTS

*Project 9.1 Lindenmayer's beautiful plants

For this project, we assume you have read Section 9.6.

Aristid Lindenmayer was specifically interested in modeling the branching behavior of plants. To accomplish this, we need to introduce two more symbols: `[` and `]`. For example, consider the following L-system:

Axiom:	`X`
Productions:	`X` → `F[-X]+X`
	`F` → `FF`
Angle:	30 degrees

These two new symbols involve the use of a simple data structure called a *stack*. A stack is simply a list in which we only append to and delete from one end. The append operation is called a *push* and the delete operation is called a *pop* (hence the name of the list `pop` method). For example, consider the following sequence of operations on a hypothetical stack object named `stack`, visualized in Figure 1.

1. `stack.push(1)`
2. `stack.push(2)`
3. `x = stack.pop()`
4. `stack.push(3)`
5. `y = stack.pop()`
6. `z = stack.pop()`

Although we implement a stack as a restricted list, it is usually visualized as a vertical stack of items in which we always push and pop items from the top. The result of the first push operation above results in the leftmost picture in Figure 1. After the second push operation, we have two numbers on the stack, with the second number on top of the first, as in the second picture in Figure 1. The third operation, a pop, removes the top item, which is then assigned to the variable name `x`. The fourth operation pushes the value 3 on the top of the stack. The fifth operation pops this value and assigns it to the variable name `y`. Finally, the bottom value is popped

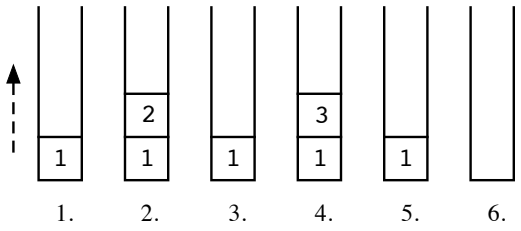


Figure 1 The results of a sequence of stack operations.

and assigned to the variable name `z`. The final values of `x`, `y`, and `z` are `2`, `3`, and `1`, respectively.

In Python, we can represent the stack as an initially empty list, implement the push operation as an `append` and implement the pop operation as a `pop` with no arguments (which defaults to deleting the last item in the list). So the equivalent sequence of Python statements is:

```
stack = [ ]       # empty stack; stack is now [ ]
stack.append(1)   # push 1;      stack is now [1]
stack.append(2)   # push 2;      stack is now [1, 2]
x = stack.pop()   # x is now 2;  stack is now [1]
stack.append(3)   # push 3;      stack is now [1, 3]
y = stack.pop()   # y is now 3;  stack is now [1]
z = stack.pop()   # z is now 1;  stack is now [ ]
```

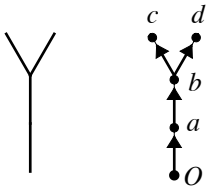
In a Lindenmayer system, the `[` symbol represents a push operation and the `]` symbol represents a pop operation. More specifically,

- `[` means "push the turtle's current position and heading on a stack," and
- `]` means "pop a position and heading from the stack and set the turtle's current position and heading to these values."

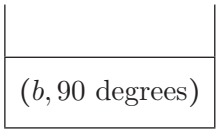
Let's now return to the Lindenmayer system above. Applying the productions of this Lindenmayer system twice results in the following string.

`X` ⇒ `F[-X]+X` ⇒ `FF[-F[-X]+X]+F[-X]+X`

The `X` symbols are used only in the derivation process and do not have any meaning for turtle graphics, so we simply skip them when we are drawing. So the string `FF[-F[-X]+X]+F[-X]+X` represents the simple "tree" below. On the left is a drawing of the tree; on the right is a schematic we will use to explain how it was drawn.

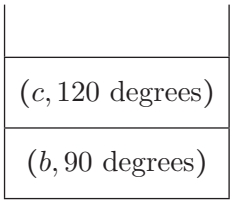


The turtle starts at the origin, marked O, with a heading of 90 degrees (north). The first two `F` symbols move the turtle forward from the origin to point a and then point b. The next symbol, `[`, means that we push the current position and heading $(b, 90$ degrees$)$ on the stack.

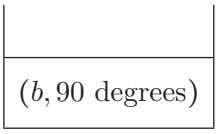


The next two symbols, `-F`, turn the turtle left 30 degrees (to a heading of 120 degrees) and move it forward, to point c. The next symbol is another `[`, which pushes the

current position and heading, $(c, 120 \text{ degrees})$, on the stack. So now the stack contains two items—$(b, 90 \text{ degrees})$ and $(c, 120 \text{ degrees})$—with the last item on top.



The next three symbols, `-X]`, turn the turtle left another 30 degrees (to a heading of 150 degrees), but then restore its heading to 120 degrees by popping $(c, 120 \text{ degrees})$ from the stack.



The next three symbols, `+X]`, turn the turtle 30 degrees to the right (to a heading of 90 degrees), but then pop $(b, 90 \text{ degrees})$ from the stack, moving the turtle back to point b, heading north.

(So, in effect, the previous six symbols, `[-X]+X` did nothing.) The next two symbols, `+F`, turn the turtle 30 degrees to the right (to a heading of 60 degrees) and move it forward to point d. Similar to before, the last six symbols, `[-X]+X`, while pushing states onto the stack, have no visible effect.

Continued applications of the productions in the L-system above will produce strings that draw the same sequence of trees that we created in Section 9.1. More involved L-systems will produce much more interesting trees. For example, the following two L-systems produce the trees in Figure 2.

Axiom:	`X`	Axiom:	`F`
Productions:	`X` → `F-[[X]+X]+F[+FX]-X`	Production:	`F` → `FF-[-F+F+F]+[+F-F-F]`
	`F` → `FF`	Angle:	22.5 degrees
Angle:	25 degrees		

Question 9.1.1 *Using an angle of* 20 *degrees, draw the figure corresponding to the string*

```
FF-[-F+F+F]+[+F-F-F]
```

(Graph paper might make this easier.)

Question 9.1.2 *Using an angle of* 30 *degrees, draw the figure corresponding to the string*

```
FF-[[FF+F]+FF+F]+FF[+FFFF+F]-FF+F
```

Figure 2 Two trees from *The Algorithmic Beauty of Plants* ([52], p. 25).

Part 1: Draw L-systems with a stack

If you have not completed Exercises 9.6.1 and 9.6.3, do that first. Then incorporate these functions, with the `derive` from Section 9.6, into a complete program. Your `main` function should call the `lsystem` function to draw a particular L-system.

Next, augment the `drawLSystem` function from Exercise 9.6.1 so that it correctly draws L-system strings containing the `[` and `]` characters. Do this by incorporating a single stack into your function, as we described above. Test your function with the three tree-like L-systems above.

Part 2: Draw L-systems recursively

The `drawLSystem` function can be implemented without an explicit stack by using recursion. Think of the `drawLSystem` function as drawing the figure corresponding to a string situated inside matching square brackets. We will pass the index of the first character after the left square bracket (`[`) as an additional parameter:

```
drawLSystem(tortoise, string, startIndex, angle, distance)
```

The function will return the index of the matching right square bracket (`]`). (We can pretend that there are imaginary square brackets around the entire string for the initial call of the function, so we initially pass in 0 for `startIndex`.) The recursive function will iterate over the indices of the characters in `string`, starting at `startIndex`. (Use a `while` loop, for reasons we will see shortly.) When it encounters a non-bracket character, it should do the same thing it did earlier. When the function encounters a left bracket, it will save the turtle's current position and heading, and then recursively call the function with `startIndex` assigned to the index of the

character after the left bracket. When this recursive call returns, the current index should be set to the index returned by the recursive call, and the function should reset the turtle's position and heading to the saved values. When it encounters a right bracket, the function will return the index of the right bracket.

For example, the string below would be processed left to right but when the first left bracket is encountered, the function would be called recursively with index 5 passed in for `startIndex`.

```
 0     5                                26
[FFFF[-FF[-F[-X]+X]+F[-X]+X]+FF[-F[-X]+X]+F[-X]+X]
      drawLSystem(..., 5, ...)
```

This recursive call will return 26, the index of the corresponding right bracket, and the `+` symbol at index 27 would be the next character processed in the loop. The function will later make two more recursive calls, marked with the two additional braces above.

Using this description, rewrite `drawLSystem` as a recursive function that does not use an explicit stack. Test your recursive function with the same tree-like L-systems, as above.

Question 9.1.3 *Why can the stack used in Part 1 be replaced by recursion in Part 2? Referring back to Figures 9.11 and 9.12, how are recursive function calls similar to pushing and popping from a stack?*

Question 9.1.4 *Use your program to draw the following additional Lindenmayer systems. For each one, set distance = 5, position = (0, – 300), heading = 90, and depth = 6.*

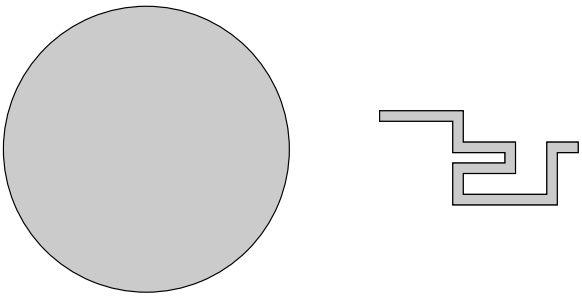
Axiom:	`X`
Productions:	`X` → `F[+X]F[-X]+X`
	`F` → `FF`
Angle:	30 degrees

Axiom:	`H`
Productions:	`H` → `HFX[+H][-H]`
	`X` → `X[-FFF][+FFF]FX`
Angle:	25.7 degrees

Project 9.2 Gerrymandering

This project assumes that you have read Section 8.3 and the part of Section 9.5 on depth-first search.

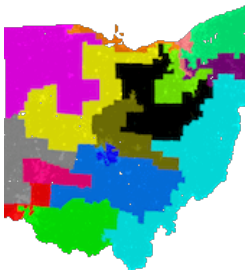
U.S. states are divided into electoral districts that each elect one person to the U.S. House of Representatives. Each district is supposed to occupy a contiguous area and represent an approximately equal number of residents. Ideally, it is also *compact*, i.e., not spread out unnecessarily. More precisely, it should have a relatively small perimeter relative to the area that it covers or, equivalently, enclose a relatively large area for a shape with its perimeter length. A perfect circle is the most compact shape, while an elongated shape, like the one on the right below, is not compact.



These two shapes actually have the same perimeter, but the shape on the right contains only about seven percent of the area of the circle on the left.

In some states, the majority political party has control over periodic redistricting. Often, the majority exploits this power by drawing district lines that favor their chances for re-election, a practice that has come to be known as *gerrymandering*. These districts often take on bizarre, non-compact shapes.

Several researchers have developed algorithms that redistrict states objectively to optimize various measures of compactness. For example, the image below on the left shows a recent district map for the state of Ohio. The image on the right shows a more compact district map.[6]



Ohio congressional districts

More compact Ohio districts

The districts on the right certainly appear to be more compact (less gerrymandered),

[6]These figures were produced by an algorithm developed by Brian Olson and retrieved from `http://bdistricting.com`

but how much better are they? In this project, we will write a program that answers this question by determining the compactness of the districts in images like these.

Part 1: Measuring compactness

The *compactness* of a region can be measured in several ways. We will consider three possibilities:

1. First, we can measure the mean of the distance between each voter and the *centroid* of the district. The centroid is the "average point," computed by averaging all of the x and y values inside the district. We might expect a gerrymandered district to have a higher mean distance than a more compact district. Since we will not actually have information fine enough to compute this value for individual voters, we will compute the average distance between the centroid and each pixel in the image of the district.
2. Second, we can measure the standard deviation of the distance between each pixel and the centroid of the district. The standard deviation measures the degree of variability from the average. Similar to above, we might expect a gerrymandered district to have higher variability in this distance. The standard deviation of a list of values (in this case, distances) is the square root of the variance. (See Exercise 7.1.10.)
3. Third, we can compare the area of the district to the area of a (perfectly compact) circle with the same perimeter. In other words, we can define

$$\text{compactness} = \frac{\text{area of district with perimeter } p}{\text{area of circle with perimeter } p}.$$

   Intuitively, a circle with a given perimeter encloses the maximum area possible for that perimeter and hence has compactness 1. A gerrymandered shape with the same perimeter encloses far less area, as we saw in the illustration above, and hence has compactness less than 1.

   Suppose that we measure a particular district and find that it has area A and perimeter p. To find the value for the denominator of our formula, we need to express the area of a circle, which is normally expressed in terms of the radius r (i.e., πr^2), in terms of p instead. To do this, recall that the formula for the perimeter of a circle is $p = 2\pi r$. Therefore, $r = p/(2\pi)$. Substituting this into the standard formula, we find that the area of a circle with perimeter p is

$$\pi r^2 = \pi \left(\frac{p}{2\pi}\right)^2 = \frac{\pi p^2}{4\pi^2} = \frac{p^2}{4\pi}.$$

   Finally, incorporating this into the formula above, we have

$$\text{compactness} = \frac{A}{\frac{p^2}{4\pi}} = \frac{4\pi A}{p^2}.$$

To compute values for the first and second compactness measures, we need a list of

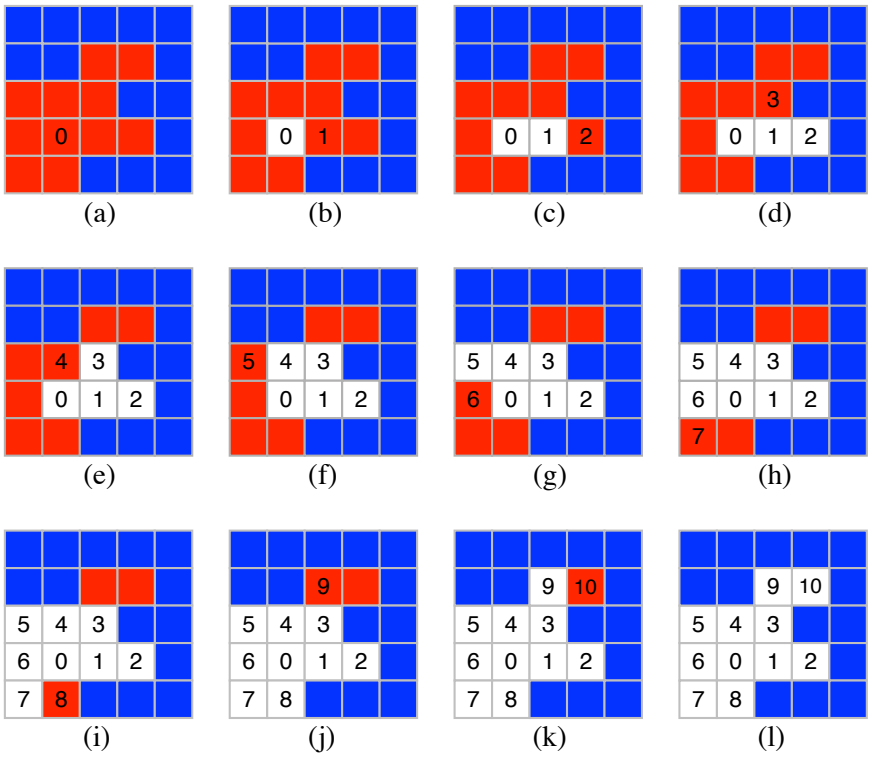


Figure 3 An example of the recursive flood fill algorithm.

the coordinates of all of the pixels in each district. With this list, we can find the centroid of the district, and then compute the mean and standard deviation of the distances from this centroid to the list of coordinates. To compute the third metric, we need to be able to determine the perimeter and area of each district.

Part 2: Measure the districts

We can accomplish all of this by using a variant of depth-first search called a *flood fill* algorithm. The idea is to start somewhere inside a district and then use DFS to explore the pixels in that district until a pixel with a different color is reached. This is illustrated in Figure 3. In this small example, the red and blue squares represent two different districts on a very small map. To explore the red district, we start at some square inside the district, in this case the square marked 0. We then explore outward using a depth-first search. As we did in Section 9.5, we will explore in clockwise order: east, south, west, north. In Figure 3(b), we mark the first square as visited by coloring it white and then recursively visit the square to the east, marked 1. After marking square 1 as visited (colored white), the algorithm explores square 2 to the east recursively, as shown in Figure 3(c). After marking square 2 as visited, the algorithm backtracks to square 1 because all four neighbors of square 2 are either a different color or have already been visited. From square 1, the algorithm next explores square 3 to the north, as shown in Figure 3(d). This process continues until

the entire red area has been visited. The numbers indicate the order in which the squares are first visited. As each square is visited for the first time, the algorithm also appends its coordinates to a list (as discussed at the end of Section 9.5). When the algorithm finishes, this list contains the coordinates of all of the squares in the red region.

Based on the `dfs` function from Section 9.5, implement this flood fill algorithm in the function

```
measureDistrict(map, x, y, color, points)
```

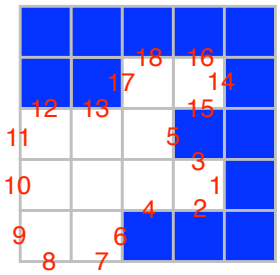
The five parameters have the following meanings:

- `map` is the name of an `Image` object (see Section 8.3) containing the district map. The flood fill algorithm will be performed on the pixels in this object rather than on a separate two-dimensional grid. There are several state maps available on the book website, all of which look similar to the maps of Ohio above.
- `x` and `y` are the coordinates of the pixel from which to begin the depth-first search.
- `color` is the color of the district being measured. This is used to tell whether the current pixel is in the desired region. You may notice that the colors of the pixels in each district are not entirely uniform across the district. (The different shades represent different population densities.) Therefore, the algorithm cannot simply check whether the color of the current pixel is equal to `color`. Rather, it needs to check whether the color of the current pixel is *close to* `color`. Since colors are represented as three-element tuples, we can treat them as three-dimensional points and use the traditional Euclidean distance formula to determine "closeness:"

  $$\text{distance}((x_1, y_1, z_1), (x_2, y_2, z_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

  Start with a distance threshold for closeness of 100, and adjust as needed.
- `points` will be a list of coordinates of the pixels that the algorithm visited. When you call the function, initially pass in an empty list. When the function returns, this list should be populated with the coordinates in the district.

Your function should return a tuple containing the total perimeter and total area obtained from a DFS starting at (x,y). The perimeter can be obtained by counting the number of times the algorithm reaches a pixel that is outside of the region (think base case), and the area is the total number of pixels that are visited inside the region. For example, as shown below, the region from Figure 3 has perimeter 18 and area 11. The red numbers indicate the order in which the flood fill algorithm will count each border.



Given these measurements, the compactness of this region is

$$\frac{4\pi \cdot 11}{18^2} \approx 0.4266.$$

The value of `points` after calling the function on this example would be

```
[(3, 1), (3, 2), (3, 3), (2, 2), (2, 1), (2, 0), (3, 0),
 (4, 0), (4, 1), (1, 2), (1, 3)]
```

The centroid of these points, derived by the averaging the x and y coordinates, is $(28/11, 15/11) \approx (2.54, 1.36)$. Then the mean distance to the centroid is approximately 1.35 and the standard deviation is approximately 0.54.

Part 3: Compare district maps

To compute the average compactness metrics for a particular map, write a function

```
compactness(imageName, districts)
```

that computes the three compactness measurements that we discussed above for the district map with file name `imageName`. You can find maps for several states on the book website. The second parameter `districts` will contain a list of starting coordinates (two-element tuples) for the districts on the map. These are also available on the book website. Your function should iterate over this list of tuples, and call your `measureDistrict` function with `x` and `y` set to the coordinates in each one. The function should return a three-element tuple containing the average value, over all of the districts, for each of the three metrics. To make sure your flood fill is working properly, it will also be helpful to display the map (using the `show` method of the `Image` class) and update it (using the `update` method of the `Image` class) in each iteration. You should see the districts colored white, one by one.

For at least three states, compare the existing district map and the more compact district map, using the three compactness measures. What do you notice?

To drive your program, write a `main` function that calls the `compactness` function with a particular map, and then reports the results. As always, think carefully about the design of your program and what additional functions might be helpful.

Technical notes

1. The images supplied on the book website have low resolution to keep the depth of the recursive calls in check. As a result, your compactness results will only

be a rough approximation. Also, shrinking the image sizes caused some of the boundaries between districts to become "fuzzy." As a result, you will see some unvisited pixels along these boundaries when the flood fill algorithm is complete.

2. Depending on your particular Python installation, the depth of recursion necessary to analyze some of these maps may exceed the maximum allowed. To increase the allowed recursion depth, you can call the `sys.setrecursionlimit` function at the top of your program. For example,

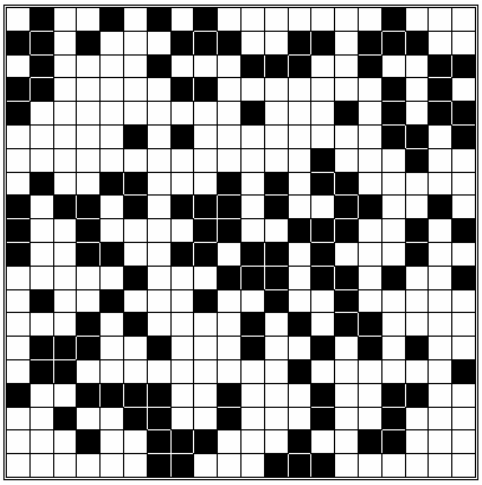
   ```
   import sys
   sys.setrecursionlimit(10000)
   ```

   However, set this value carefully. Use the smallest value that works. Setting the maximum recursion depth too high may crash Python on your computer! If you cannot find a value that works on your computer, try shrinking the image file instead (and scaling the starting coordinates appropriately).

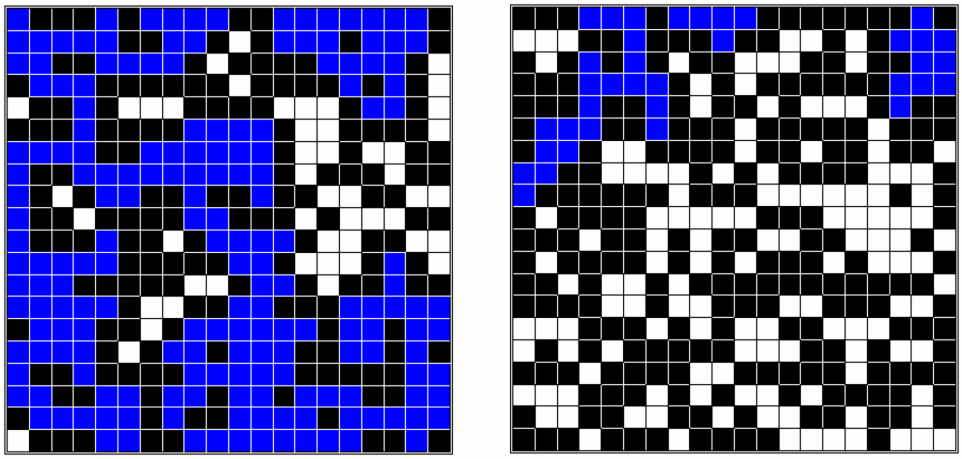
Project 9.3 Percolation

This project assumes that you are familiar with two-dimensional grids from Section 8.2 and have read the part of Section 9.5 on depth-first search.

Suppose we have a grid of squares called *sites*. A site can either be open (white) or blocked (black).



Now imagine that we pour a liquid uniformly over the top of the grid. The liquid will fill the open sites at the top and percolate to connected open sites until the liquid fills all of the open sites that are reachable from an open site at the top. We say that the grid *percolates* if at least one open site in the bottom row is full at the end. For example, the grid below on the left percolates, while the grid on the right does not.



This system can be used to model a variety of naturally occurring phenomena. Most obviously, it can model a porous rock being saturated with water. Similarly, it can model an oil (or natural gas) field; in this case, the top of the grid represents the oil underground and percolation represents the oil reaching the surface. Percolation

systems can also be used to model the flow of current through a network of transistors, whether a material conducts electricity, the spread of disease or forest fires, and even evolution.

We can represent how "porous" a grid is by a *vacancy probability*, the probability that any particular site is open. For a variety of applications, scientists are interested in knowing the probability that a grid with a particular vacancy probability will percolate. In other words, we would like to know the percolation probability for any vacancy probability p. Despite decades of research, there is no known mathematical solution to this problem. Therefore, we will use a Monte Carlo simulation to estimate it.

Recall from Chapter 5 that a Monte Carlo simulation flips coins (metaphorically speaking) at each step in a computation. For example, to estimate the distance traveled by a random walk, we performed many random walks and took the average final distance from the origin. In this problem, for any given vacancy probability p, we will create many random grids and then test whether they percolate. By computing the number that do percolate divided by the total number of trials, we will estimate the percolation probability for vacancy probability p.

Part 1: Does it percolate?

To decide whether a grid percolates, we can use a depth-first search. Recall from Section 9.5 that the depth-first search algorithm completely explores a space by first exploring as far away as possible from the source. Then it *backtracks* to follow paths that branch off. To decide whether a given grid percolates, we must do a depth-first search from each of the open sites in the top row. Once this is done, we simply look at whether any site in the bottom row has been visited. If so, the system percolates. Write a function

```
percolates(grid, draw)
```

that decides whether a given grid percolates. The second parameter is a Boolean that indicates whether the grid should also be drawn using turtle graphics. There is a skeleton program on the book website in which the drawing code has already been written. Notice that some of the functions include a Boolean parameter that indicates whether the percolation should be visualized.

Part 2: Find the percolation probability

For any particular vacancy probability p, we can design a Monte Carlo simulation to estimate the percolation probability:

1. Create a random grid in which, with vacancy probability p, any particular site is open.
2. Test to see if this grid percolates.
3. Repeat steps 1 and 2 a large number of times (say, 10,000), keeping track of the number of grids that percolate.

4. Divide the number that percolate by the total number of trials. This is your estimated percolation probability.

Implement this algorithm with a function

```
percMonteCarlo(rows, columns, p, trials)
```

Part 3: When is a grid likely to percolate?

For what vacancy probability does the percolation probability reach at least 1/2? In other words, what must the vacancy probability be for a system to be more likely than not to percolate?

To answer this question, also write a function

```
percPlot(minP, maxP, stepP, trials)
```

that plots vacancy probability on the x axis and percolation probability on the y axis for vacancy probabilities `minP`, `minP + stepP`, ..., `maxP`. Each percolation probability should be derived from a Monte Carlo simulation with the given number of trials.

You should discover a *phase transition*: if the vacancy probability is less than a particular *threshold* value, the system almost certainly does not percolate; but if the vacancy probability is greater than this threshold value, the system almost certainly *does* percolate. What is this threshold value?