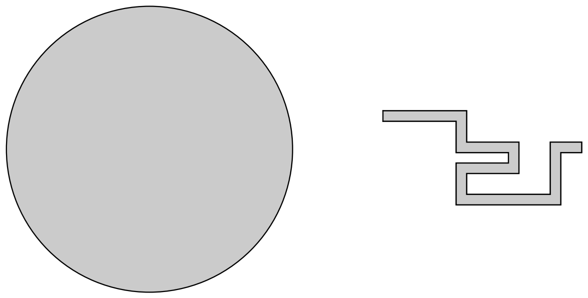
Project 9.2 Gerrymandering

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

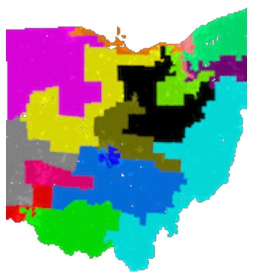
U.S. states are divided into electoral districts that each elect one person to the U.S. House of Representatives. Each district is supposed to occupy a contiguous area and represent an approximately equal number of residents. Ideally, it is also *compact*, i.e., not spread out unnecessarily. More precisely, 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.



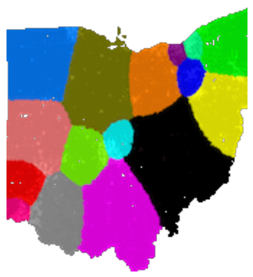
These two shapes actually have the same perimeter, but the shape on the right contains only about seven percent of the area of the circle on the left.

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

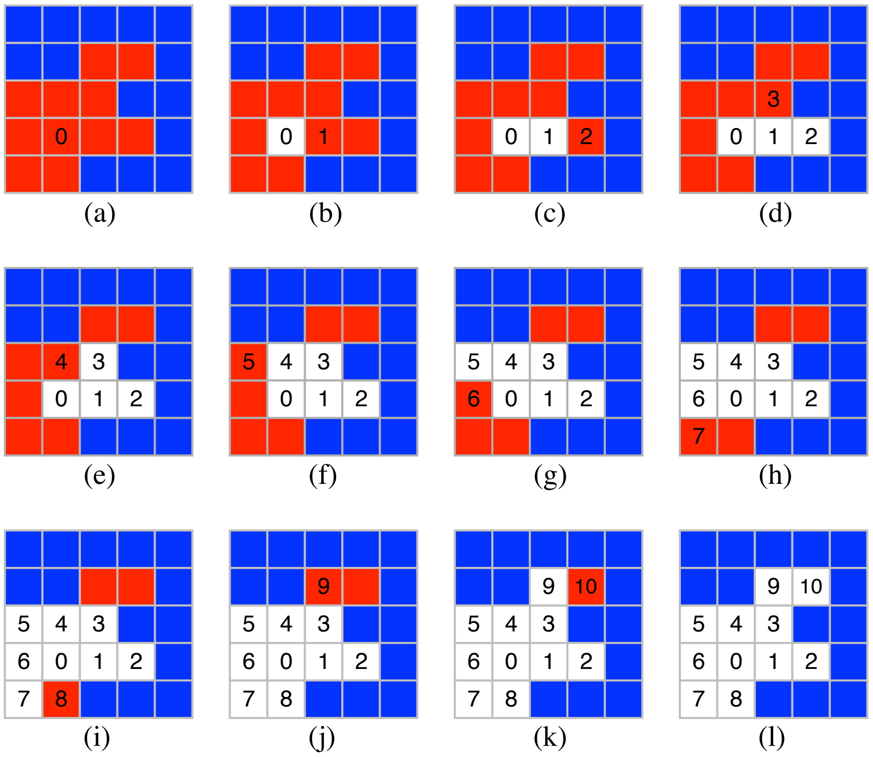


Figure 3 An example of the recursive flood fill algorithm.

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)
```

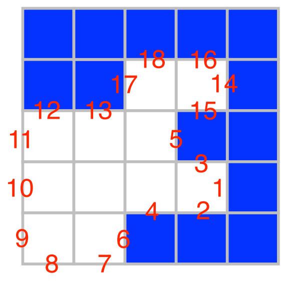
The five parameters have the following meanings:

- `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).