

# RECURSION

## Part 2

Recursion Case Study

# Recursive Thinking

## Section 5.1

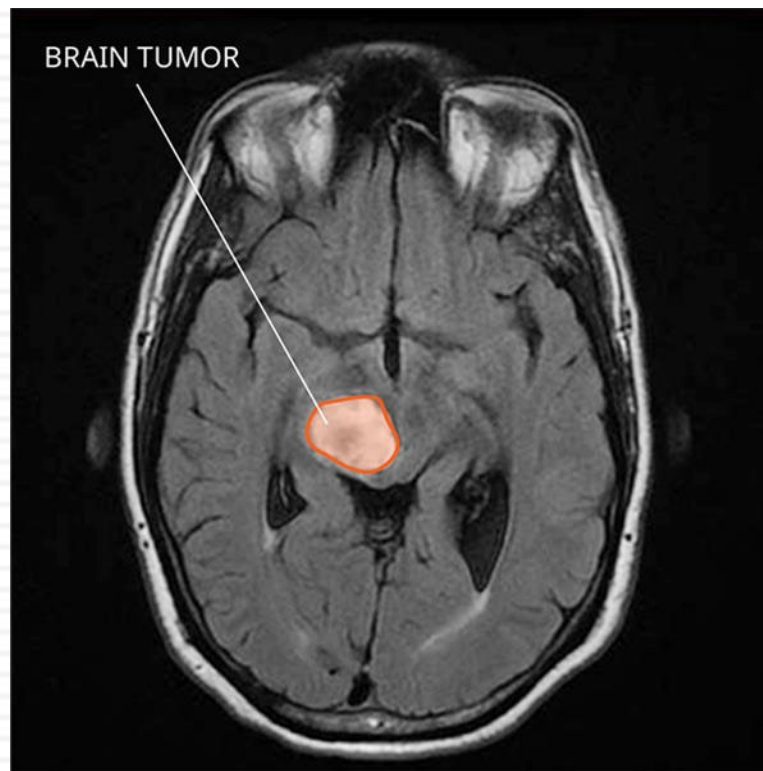
# Steps to Design a Recursive Algorithm

- Identify the base case(s) and solve it/them directly
- Devise a strategy to reduce the problem to smaller versions of itself while making progress toward the base case
- Combine the solutions to the smaller problems to solve the larger problem

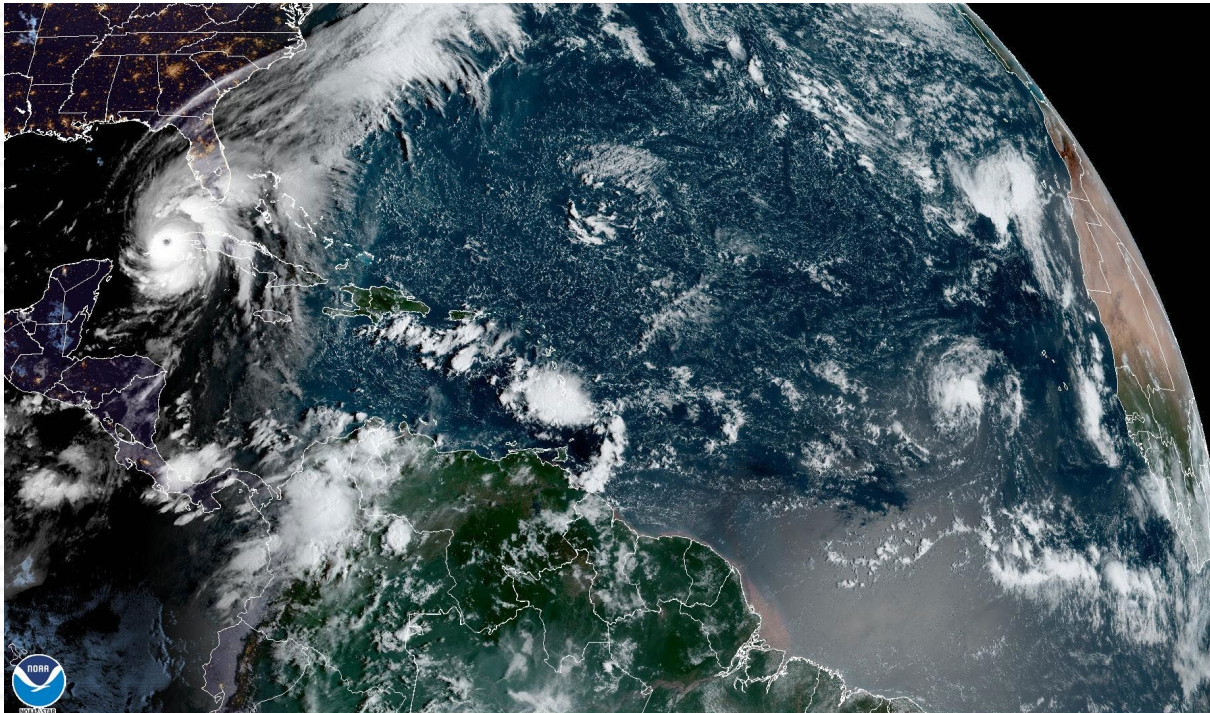
# Problem Solving with Recursion

## Section 5.5

# Image Processing



# Image Processing



27 Sep 2022 11:50Z - NOAA/NESDIS/STAR - GOES-East - GEOCOLOR Composite TAW

# Counting Cells in a Blob

- Consider how we might process an image that is presented as a two-dimensional array of color values
- Information in the image may come from
  - ▣ an X-ray
  - ▣ an MRI
  - ▣ satellite imagery
  - ▣ etc.
- The goal is to **determine the size of any area** in the image that is considered abnormal because of its color values

# Counting Cells in a Blob—the Problem

- Given a two-dimensional grid of cells, each cell contains either a normal background color or a second color, which indicates the presence of an abnormality
- A *blob* is a collection of contiguous abnormal cells
- A user will enter the  $x, y$  coordinates of a cell in the blob, and the program will determine the count of all cells in that blob



# Counting Cells in a Blob—Analysis

- Problem Inputs

- ▣ the two-dimensional grid of cells
- ▣ the coordinates of a cell in a blob

- Problem Outputs

- ▣ the count of cells in the blob

# Counting Cells in a Blob—Design

Method	Behavior
<code>void recolor(int x, int y, Color aColor)</code>	Resets the color of the cell at position (x, y) to aColor.
<code>Color getColor(int x, int y)</code>	Retrieves the color of the cell at position (x, y).
<code>int getNRows()</code>	Returns the number of cells in the y-axis.
<code>int getNCols()</code>	Returns the number of cells in the x-axis.

Method	Behavior
<code>int countCells(int x, int y)</code>	Returns the number of cells in the blob at (x, y).

# Counting Cells in a Blob—Design (cont.)

## Algorithm for `countCells(x, y)`

```
if the cell at (x, y) is outside the grid
    the result is 0
else if the color of the cell at (x, y) is not the abnormal color
    the result is 0
else
    set the color of the cell at (x, y) to a temporary color
    the result is 1 plus the number of cells in each piece of the blob that
    includes a nearest neighbor
```


# Counting Cells in a Blob—Implementation

## (cont.)

```
/** Finds the number of cells in the blob at (x,y).
    pre: Abnormal cells are in ABNORMAL color;
        Other cells are in BACKGROUND color.
    post: All cells in the blob are in the TEMPORARY color.
    @param x The x-coordinate of a blob cell
    @param y The y-coordinate of a blob cell
    @return The number of cells in the blob that contains (x, y)
*/
public int countCells(int x, int y) {
    int result;

    if (x < 0 || x >= grid.getNCols()
        || y < 0 || y >= grid.getNRows())
        return 0;
    else if (!grid.getColor(x, y).equals(ABNORMAL))
        return 0;
    else {
        grid.recolor(x, y, TEMPORARY);
        return 1
            + countCells(x - 1, y + 1) + countCells(x, y + 1)
            + countCells(x + 1, y + 1) + countCells(x - 1, y)
            + countCells(x + 1, y) + countCells(x - 1, y - 1)
            + countCells(x, y - 1) + countCells(x + 1, y - 1);
    }
}
```


# Counting Cells in a Blob—Testing

 - □ ×

Toggle a button to change its color --  
When done, press SOLVE.  
Blob count will start at the last button pressed

0,0	1,0	2,0	3,0	4,0	5,0
0,1	1,1	2,1	3,1	4,1	5,1
0,2	1,2	2,2	3,2	4,2	5,2
0,3	1,3	2,3	3,3	4,3	5,3

SOLVE

 - □ ×

Toggle a button to change its color --  
When done, press SOLVE.  
Blob count will start at the last button pressed

0,0	1,0	2,0			
0,1	1,1	2,1	3,1		5,1
0,2	1,2	2,2		4,2	
0,3	1,3	2,3	3,3		5,3

SOLVE

# Counting Cells in a Blob—Testing (cont.)

- Verify that the code works for the following cases:
  - ▣ A starting cell that is on the **edge** of the grid
  - ▣ A starting cell that has **no neighboring abnormal** cells
  - ▣ A starting cell whose **only abnormal** neighbor cells are **diagonally connected** to it
  - ▣ A "**bull's-eye**": a starting cell whose neighbors are all normal but their neighbors are abnormal
  - ▣ A starting cell that is **normal**
  - ▣ A grid that contains **all abnormal cells**
  - ▣ A grid that contains **all normal cells**