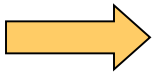
Recursion

Recursion

- Recursion is a fundamental programming technique that can provide an elegant solution certain kinds of problems
- We will focus on:
  - thinking in a recursive manner
  - programming in a recursive manner
  - the correct use of recursion
  - recursion examples

Outline



Recursive Thinking

Recursive Programming

Using Recursion

Recursion in Graphics

Recursive Thinking

- A recursive definition is one which uses the word or concept being defined in the definition itself
- When defining an English word, a recursive definition is often not helpful
- But in other situations, a recursive definition can be an appropriate way to express a concept
- Before applying recursion to programming, it is best to practice thinking recursively

Recursive Definitions

- Consider the following list of numbers:
- 24, 88, 40, 37
- Such a list can be defined as follows:

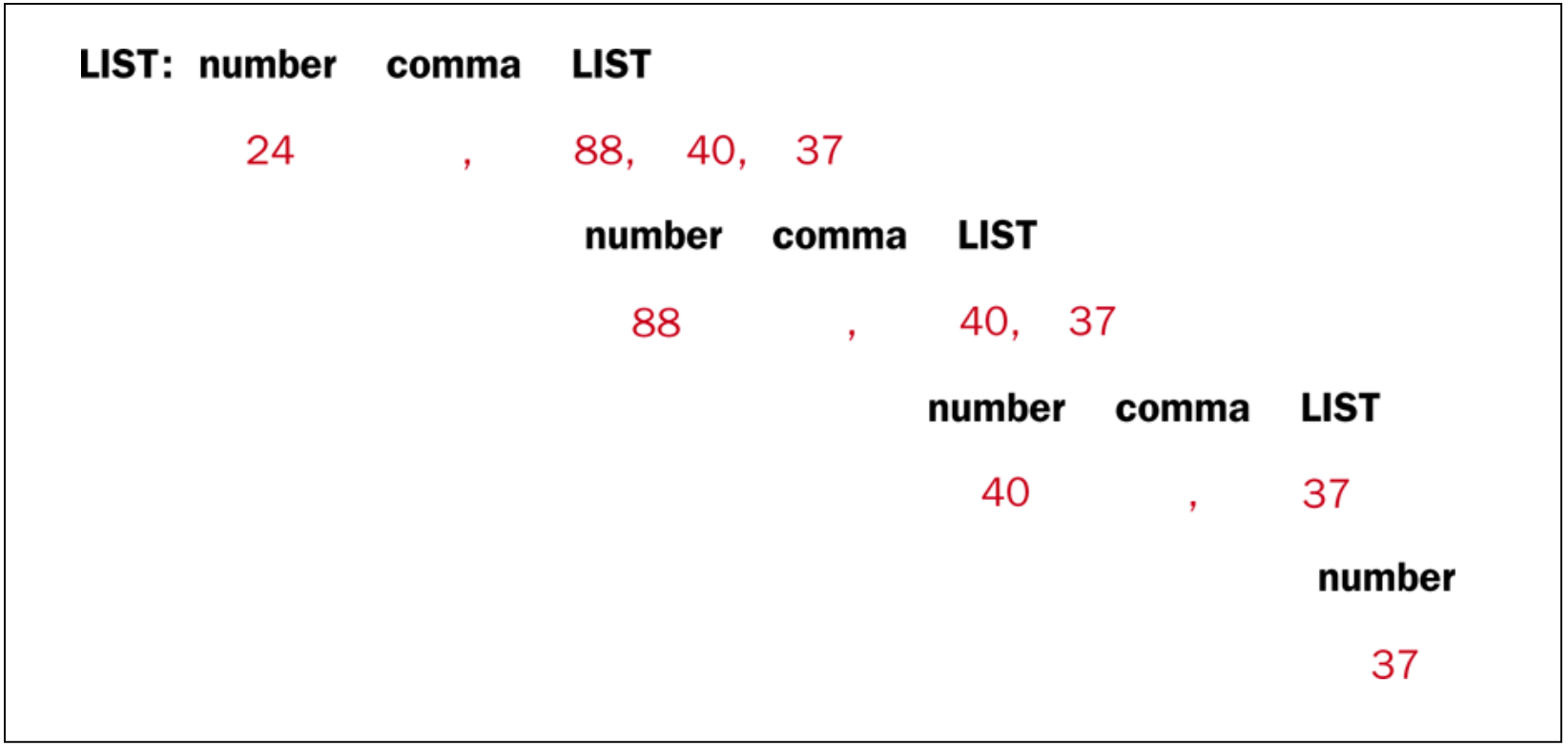
  A List is a: number

  or a: number comma List

- That is, a List is defined to be a single number, or a number followed by a comma followed by a List
- The concept of a List is used to define itself

Recursive Definitions

- The recursive part of the LIST definition is used several times, terminating with the non-recursive part:



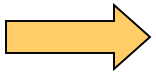
Infinite Recursion

- All recursive definitions have to have a non-recursive part called the **base case**

- If they didn't, there would be no way to terminate the recursive path
- Such a definition would cause infinite recursion
- This problem is similar to an infinite loop, but the non-terminating "loop" is part of the definition itself
- **You must always have some base case which can be solved without recursion**

Outline

Recursive Thinking



Recursive Programming

Using Recursion

Recursion in Graphics

Recursive Programming

- **A recursive method is a method that invokes itself**
- A recursive method must be structured to **handle both the base case and the recursive case**
- Each call to the method sets up a new execution environment, with new parameters and local variables
- As with any method call, when the method completes, control returns to the method that invoked it (which may be an earlier invocation of itself)

Sum of 1 to N

- Consider the problem of computing the sum of all the numbers between 1 and any positive integer N
- This problem can be recursively defined as:

$$\sum_{i=1}^{N} i = N + \sum_{i=1}^{N-1} i = N + N-1 + \sum_{i=1}^{N-2} i$$

$$= N + N-1 + N-2 + \sum_{i=1}^{N-3} i$$

$$\vdots$$

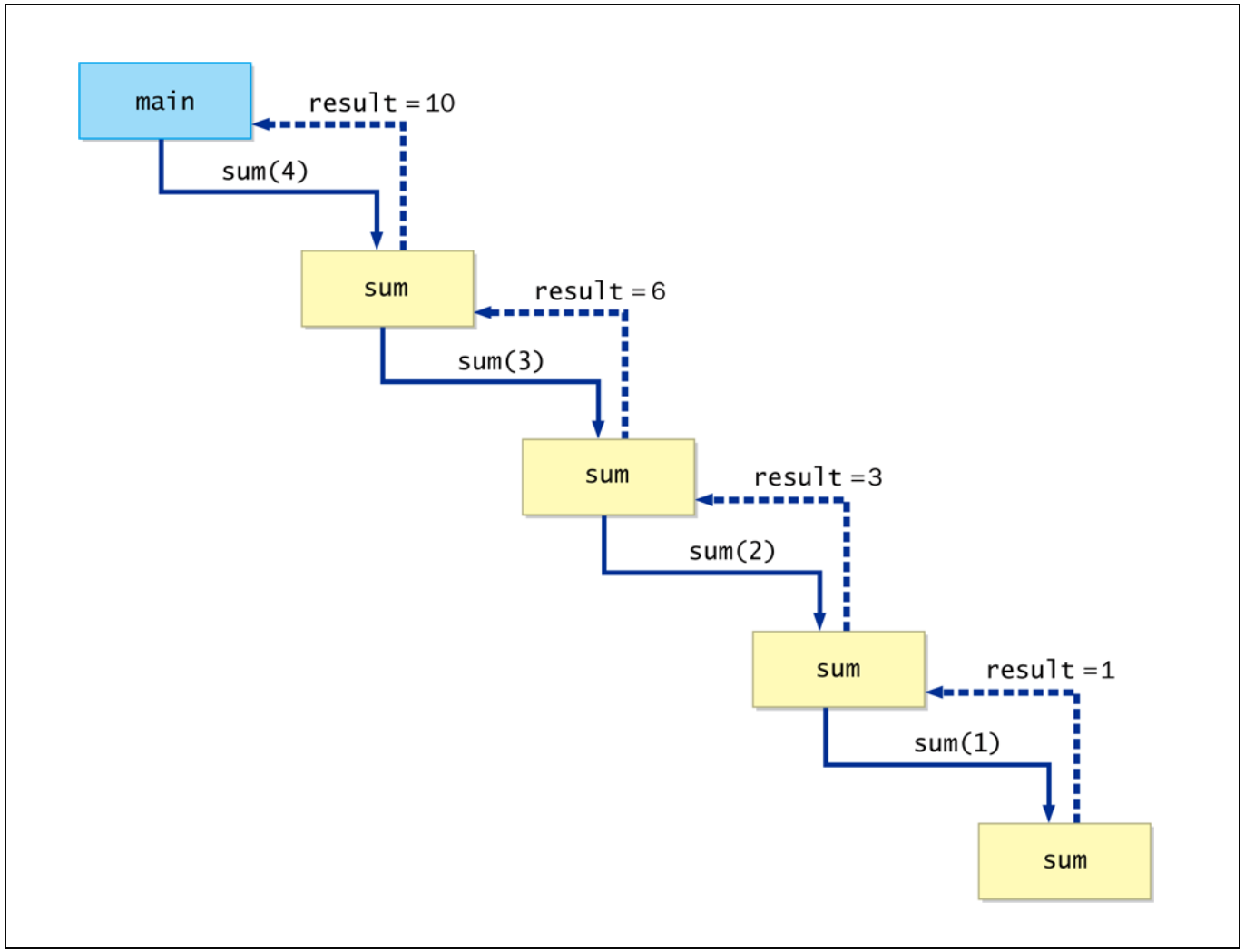
$$= N + N-1 + N-2 + \cdots + 2 + 1$$

Sum of 1 to N

- The summation could be implemented recursively as follows:

```
// This method returns the sum of 1 to num
public int sum (int num)
{
   int result;

   if (num == 1)
      result = 1;
   else
      result = num + sum (n-1);

   return result;
}
```

Sum of 1 to N


main
result = 10
sum(4)
sum
result = 6
sum(3)
sum
result = 3
sum(2)
sum
result = 1
sum(1)
sum

Recursive Programming

- **Note that just because we can use recursion to solve a problem, doesn't mean we should**

- We usually would not use recursion to solve the summation problem, because the iterative version is easier to understand
- However, for some problems, recursion provides an elegant solution, often cleaner than an iterative version
- You must carefully decide whether recursion is the correct technique for any problem

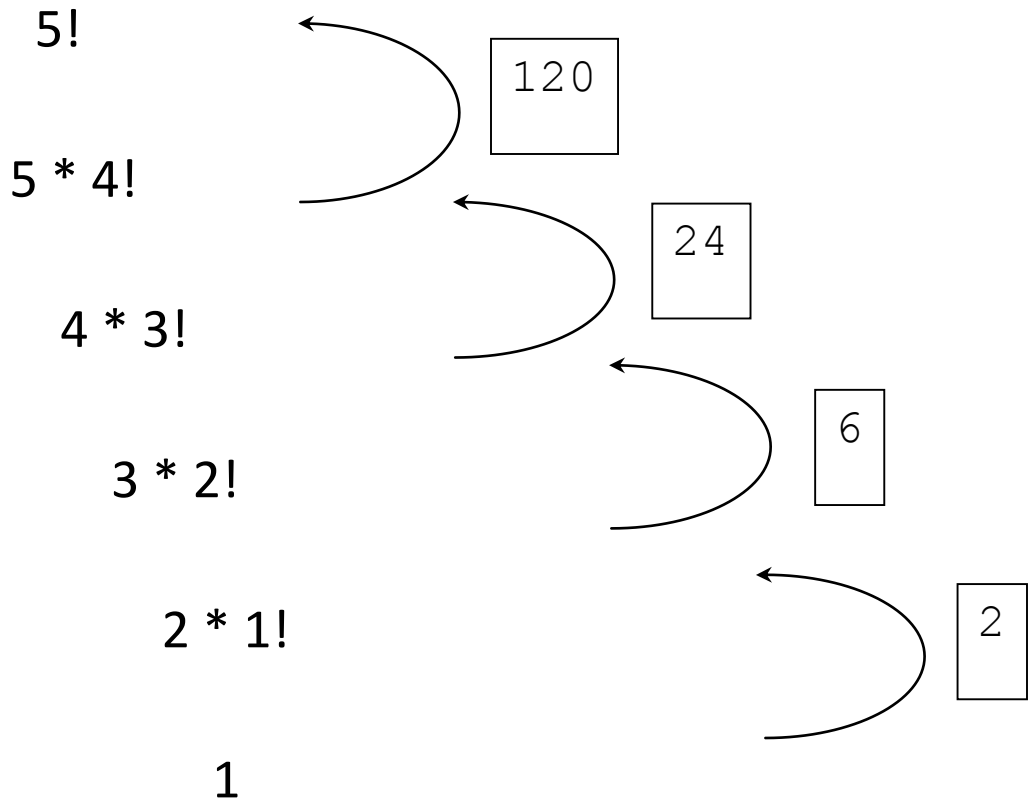
Recursive Factorial

- N!
- For any positive integer N, is defined to be the product of all integers between 1 and N inclusive
- This definition can be expressed recursively as:

  $1! = 1$

  $N! = N * (N-1)!$
- A factorial is defined in terms of another factorial
- Eventually, the base case of 1! is reached

Recursive Factorial



Recursive Factorial

```
public class FindFactorialRecursive
{
        public static void main (String args[])
        {
                System.out.println ("3! = " + factorial(3));
        }

        public static int factorial(int number)
        {
                if (( number == 1) || (number == 0))
                        return 1;
                else
                        return (number * factorial(number-1));
        }
}
```

Base Case.

Making progress

Seeing the Pattern

- Seeing the pattern in the factorial example is difficult at first.
- But, once you see the pattern, you can apply this pattern to create a recursive solution to the problem.

- **Divide a problem up into:**
  - What we know (call this the base case)
  - Making progress towards the base
    - Each step resembles original problem
    - The method launches a new copy of itself (recursion step) to make the progress.

Visualizing Recursion

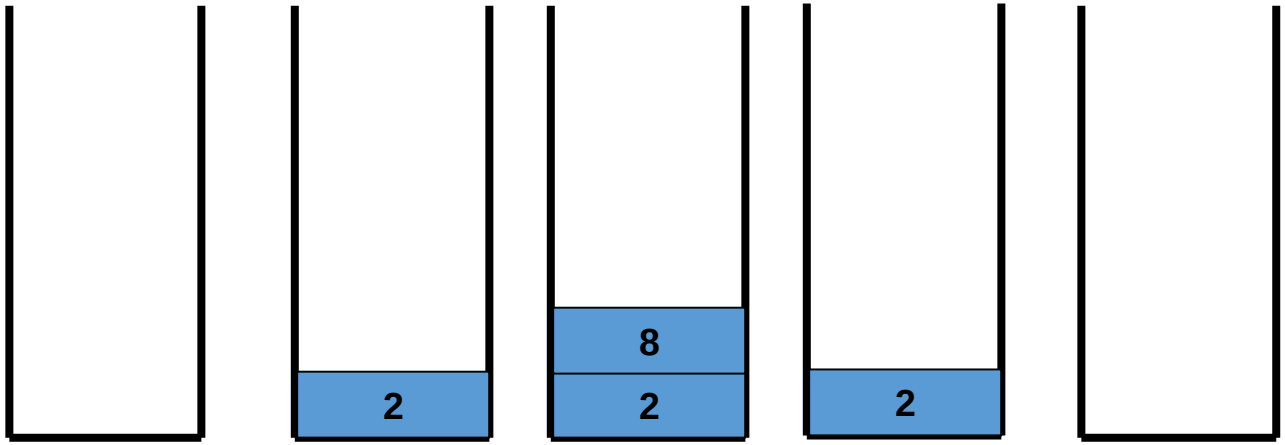
- To understand how recursion works, it helps to visualize what's going on.

- To help visualize, we will use a common concept called the **Stack**.
- A stack basically operates like a container of trays in a cafeteria. It has only two operations:
  - **Push:** you can push something onto the stack.
  - **Pop:** you can pop something off the top of the stack.

- Let's see an example stack in action.

Stacks

The diagram below shows a stack over time.
We perform two pushes and one pop.



Time: 0
Empty Stack

Time 1:
Push “2”

Time 2:
Push “8”

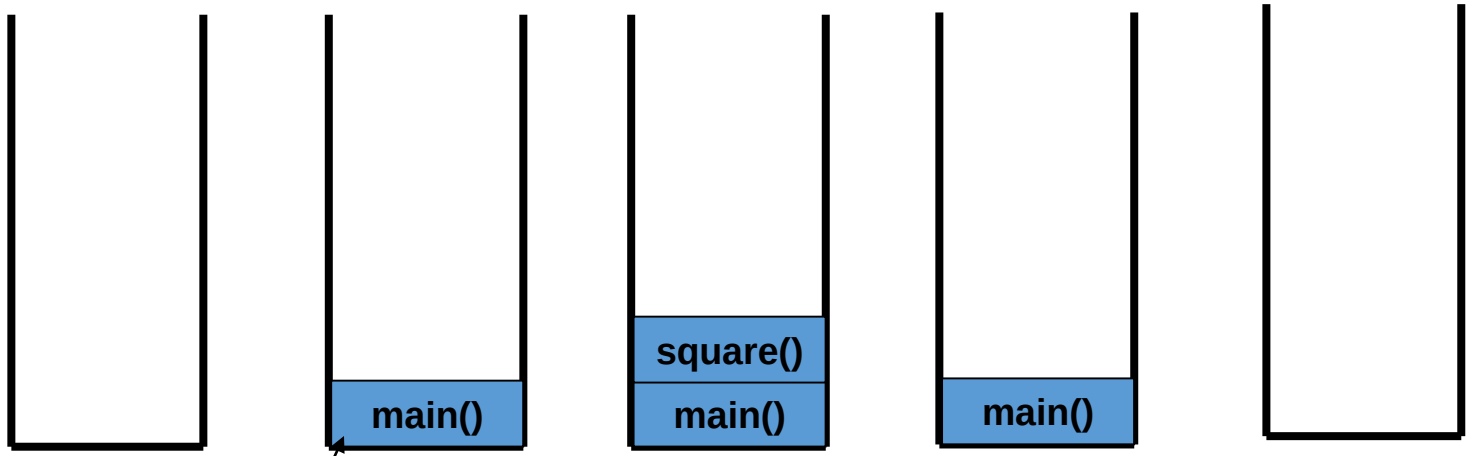
Time 3:
Pop: Gets 8

Time 4:
Pop: Gets 2

Stacks and Methods

- When you run a program, the computer creates a stack for you.
- Each time you invoke a method, the method is placed on top of the stack.
- When the method returns or exits, the method is popped off the stack.
- The diagram on the next page shows a sample stack for a simple Java program.

Stacks and Methods



Time: 0
Empty Stack

Time 1:
Push: main()

Time 2:
Push: square()

Time 3:
Pop: square()
returns a value.
method exits.

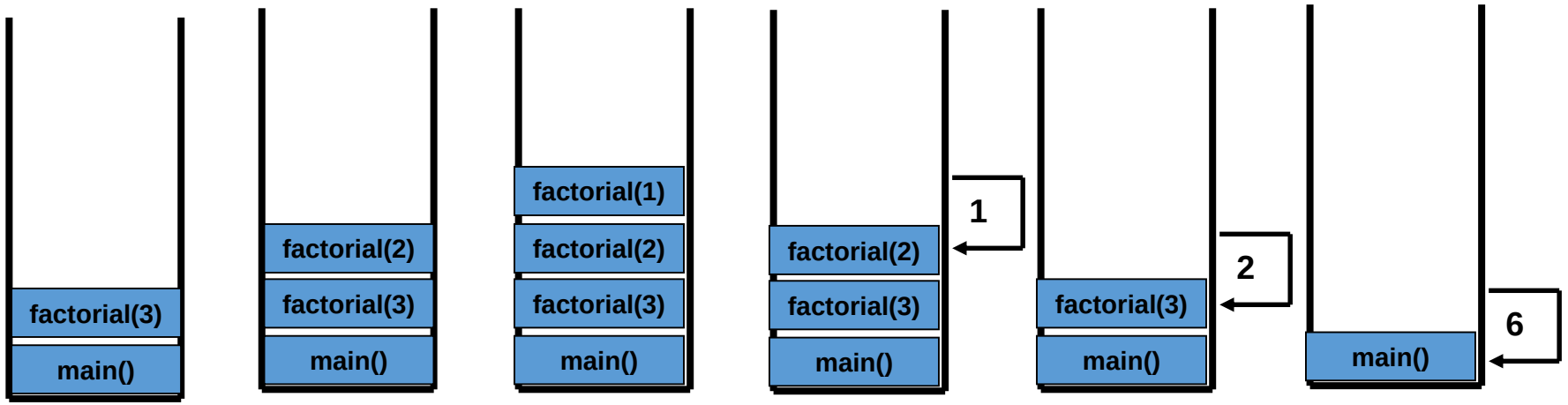
Time 4:
Pop: main()
returns a value.
method exits.

This is called an activation record or stack frame.

Stacks and Recursion

- Each time a method is called, you push the method on the stack.
- Each time the method returns or exits, you pop the method off the stack.
- **If a method calls itself recursively, you just push another copy of the method onto the stack.**
- We therefore have a simple way to visualize how recursion really works.

Finding the factorial of 3



Time 2:
Push:
`factorial(3)`

Time 3:
Push:
`factorial(2)`

Time 4:
Push:
`factorial(1)`

Time 5:
Pop:
`factorial(1)`
`returns 1.`

Time 6:
Pop:
`factorial(2)`
`returns 2.`

Time 7:
Pop:
`factorial(3)`
`returns 6.`

Inside `factorial(3)`:

```
if (number <= 1)
   return 1;
else
return (3*factorial(2));
```

Inside factorial(2):

```
if (number <= 1)
return 1;
else
return (2*factorial(1));
```

Inside factorial(1):

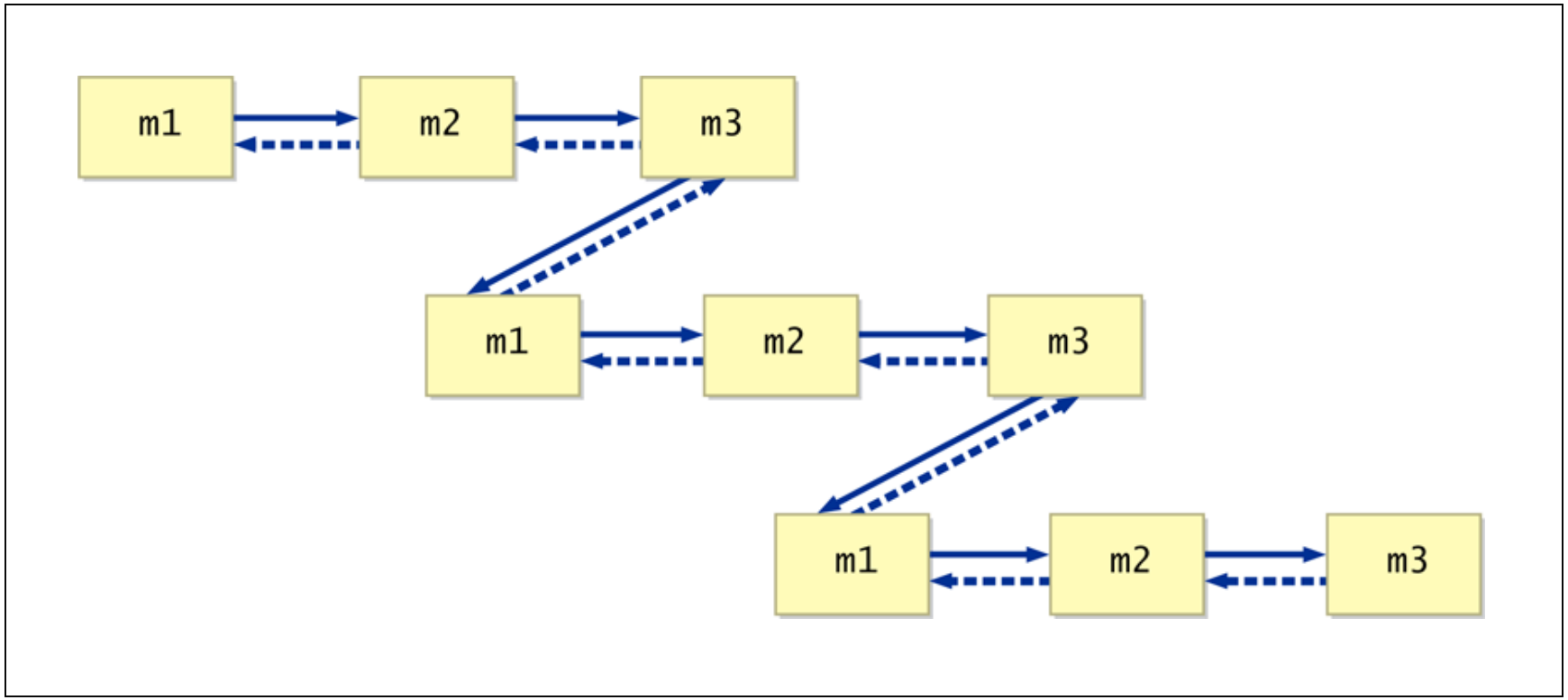
```
if (number <= 1) return 1;
else
return (1*factorial(0));
```

Indirect Recursion

- A method invoking itself is considered to be direct recursion
- A method could invoke another method, which invokes another, etc., until eventually the original method is invoked again

- For example, method m1 could invoke m2, which invokes m3, which in turn invokes m1 again
- This is called indirect recursion, and requires all the same care as direct recursion
- It is often more difficult to trace and debug

Indirect Recursion



Thinking Recursively

© Nikada/iStockphoto.

Thinking recursively is easy if you can recognize a subtask that is similar to the original task.

- Problem: test whether a sentence is a palindrome
- Palindrome: a string that is equal to itself when you reverse all characters
  - A man, a plan, a canal – Panama!
  - Go hang a salami, I'm a lasagna hog
  - Madam, I'm Adam

Implement `isPalindrome` Method: How To 12.1

```
public class Sentence
{
   private String text;
   /**
      Constructs a sentence.
      @param aText a string containing all characters of the sentence
   */
   public Sentence(String aText)
   {
       text = aText;
   }
   /**
      Tests whether this sentence is a palindrome.
      @return true if this sentence is a palindrome, false otherwise
   */
   public boolean isPalindrome()
   {
      . . .
   }
}
```

Thinking Recursively: How To 12.1

1. Consider various ways to simplify inputs. Here are several possibilities:

- Remove the first character.
- Remove the last character.
- Remove both the first and last characters.
- Remove a character from the middle.
- Cut the string into two halves.

Thinking Recursively: How To 12.1

2. Combine solutions with simpler inputs into a solution of the original problem.

- Most promising simplification: Remove first and last characters
  "adam, I'm Ada", is a palindrome too!
- Thus, a word is a palindrome if
  - The first and last letters match, and
  - Word obtained by removing the first and last letters is a palindrome
- What if first or last character is not a letter? Ignore it.
  - If the first and last characters are letters, check whether they match;
    if so, remove both and test shorter string
  - If last character isn't a letter, remove it and test shorter string
  - If first character isn't a letter, remove it and test shorter string

Thinking Recursively: How To 12.1

3. Find solutions to the simplest inputs.

- Strings with two characters
  - No special case required; step two still applies
- Strings with a single character
  - They are palindromes
- The empty string
  - It is a palindrome

Thinking Recursively: How To 12.1

4. Implement the solution by combining the simple cases and the reduction step:

```
public boolean isPalindrome()
{
   int length = text.length();
   // Separate case for shortest strings.
   if (length <= 1) { return true; }
   // Get first and last characters, converted to lowercase.
   char first = Character.toLowerCase(text.charAt(0));
   char last = Character.toLowerCase(text.charAt(length - 1));
```

Continued

Thinking Recursively: How To 12.1

```
if (Character.isLetter(first) && Character.isLetter(last))
   {
      // Both are letters.
      if (first == last)
      {
         // Remove both first and last character.
         Sentence shorter =
            new Sentence(text.substring(1, length - 1));
         return shorter.isPalindrome();
      }
      else
      {
         return false;
      }
   }
```

Continued

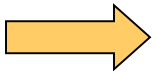
Thinking Recursively: How To 12.1

```
else if (!Character.isLetter(last))
   {
      // Remove last character.
      Sentence shorter = new Sentence(text.substring(0, length - 1));
      return shorter.isPalindrome();
   }
   else
   {
      // Remove first character.
      Sentence shorter = new Sentence(text.substring(1));
      return shorter.isPalindrome();
   }
}
```

Outline

Recursive Thinking

Recursive Programming



Using Recursion

Recursion in Graphics

Maze Traversal

```
1110110001111
1011101111001
0000101010100
1110111010111
1010000111001
1011111101111
1000000000000
1111111111111
```

Maze Traversal

- We can use recursion to find a path through a maze
- From each location, we can search in each direction
- The recursive calls keep track of the path through the maze
- See MazeSearch.java
- See Maze.java

```
//********************************************************************
//  MazeSearch.java       Author: Lewis/Loftus
//
//  Demonstrates recursion.
//********************************************************************

public class MazeSearch
{
   //-----------------------------------------------------------------
   //  Creates a new maze, prints its original form, attempts to
   //  solve it, and prints out its final form.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Maze labyrinth = new Maze();

      System.out.println (labyrinth);

      if (labyrinth.traverse (0, 0))
         System.out.println ("The maze was successfully traversed!");
      else
         System.out.println ("There is no possible path.");

      System.out.println (labyrinth);
   }
}
```

```
//**********                                                    ************
//  MazeSear
//
//  Demonstr                                                    ************
//**********

public class
{
   //-------                                                    -------------
   //  Creat                                                    mpts to
   //  solve
   //-------                                                    -------------
   public st
   {
      Maze l

      System

      if (la
         Sys                                                    raversed!");
      else
         Sys                                                    );

      System
   }
}
```

Output

```
1110110001111
1011101111001
0000101010100
1110111010111
1010000111001
1011111101111
1000000000000
1111111111111

The maze was successfully traversed!

7770110001111
3077707771001
0000707070300
7770777070333
7070000773003
7077777703333
7000000000000
7777777777777
```

Maze Traversal

- Write the `traverse` method
  - `private final int TRIED = 3;`
  - `private final int PATH = 7;`
- How do you define each step in the maze?
- Where do you start?
- First check the validity of the existing position
  - `Valid method`

```
// check if cell is in the bounds of the matrix
// check if cell is not blocked and not previously tried
```

- Follow this sequence:
  - Down, Right, Up, Left
- What is the base case?
  - The base case is an invalid move or reaching the final destination

```
//********************************************************************
//  Maze.java       Author: Lewis/Loftus
//
//  Represents a maze of characters. The goal is to get from the
//  top left corner to the bottom right, following a path of 1s.
//********************************************************************

public class Maze
{
   private final int TRIED = 3;
   private final int PATH = 7;

   private int[][] grid = { {1,1,1,0,1,1,0,0,0,1,1,1,1},
                            {1,0,1,1,1,0,1,1,1,1,0,0,1},
                            {0,0,0,0,1,0,1,0,1,0,1,0,0},
                            {1,1,1,0,1,1,1,0,1,0,1,1,1},
                            {1,0,1,0,0,0,0,1,1,1,0,0,1},
                            {1,0,1,1,1,1,1,1,0,1,1,1,1},
                            {1,0,0,0,0,0,0,0,0,0,0,0,0},
                            {1,1,1,1,1,1,1,1,1,1,1,1,1} };
```

continued

```
 //-----------------------------------------------------------------
//  Attempts to recursively traverse the maze. Inserts special
//  characters indicating locations that have been tried and that
//  eventually become part of the solution.
//-----------------------------------------------------------------
public boolean traverse (int row, int column)
{
        boolean done = false;

        if (valid (row, column))
        {
                grid[row][column] = TRIED;  // this cell has been tried

                if (row == grid.length-1 && column == grid[0].length-1)
                        done = true;  // the maze is solved
                else
                {
                        done = traverse (row+1, column);     // down
                        if (!done)
                                done = traverse (row, column+1);  // right
                        if (!done)
                                done = traverse (row-1, column);  // up
                        if (!done)
                                done = traverse (row, column-1);  // left
                }
```

continued

continued

```
            if (done)  // this location is part of the final path
               grid[row][column] = PATH;
         }

         return done;
      }

      //-----------------------------------------------------------------
      //  Determines if a specific location is valid.
      //-----------------------------------------------------------------
      private boolean valid (int row, int column)
      {
         boolean result = false;

         // check if cell is in the bounds of the matrix
         if (row >= 0 && row < grid.length &&
             column >= 0 && column < grid[row].length)

            //  check if cell is not blocked and not previously tried
            if (grid[row][column] == 1)
               result = true;

         return result;
      }
```

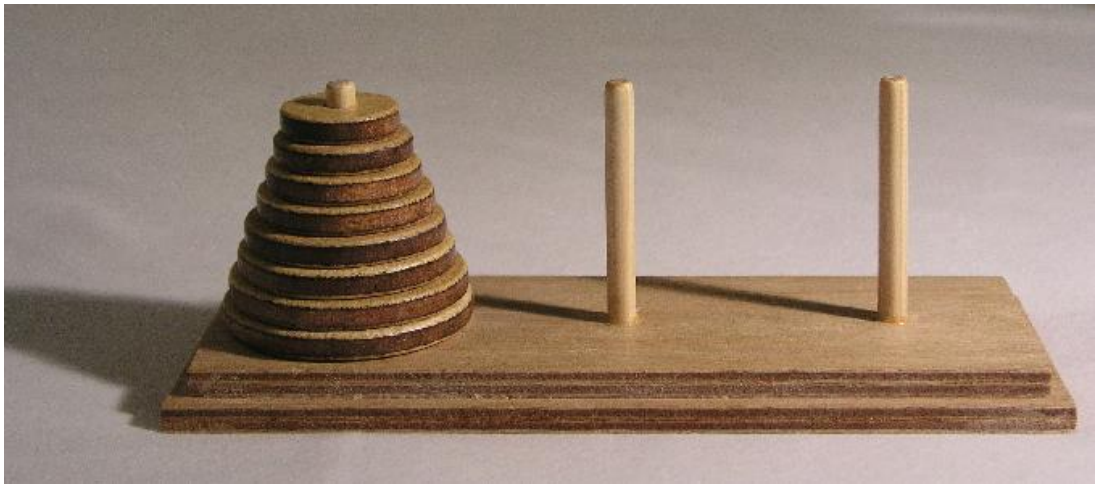
continued

continued

```
    //-----------------------------------------------------------------
    //  Returns the maze as a string.
    //-----------------------------------------------------------------
    public String toString ()
    {
        String result = "\n";

        for (int row=0; row < grid.length; row++)
        {
            for (int column=0; column < grid[row].length; column++)
                result += grid[row][column] + "";
            result += "\n";
        }

        return result;
    }
}
```

Towers of Hanoi

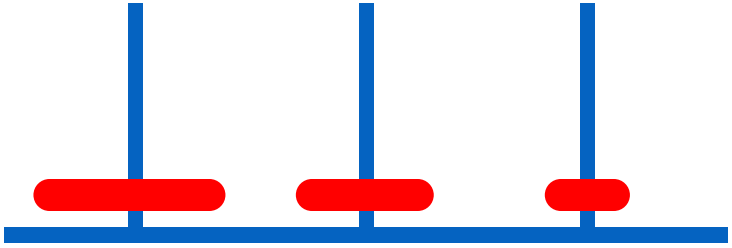
- The Towers of Hanoi is a puzzle made up of three vertical pegs and several disks that slide onto the pegs
- The disks are of varying size, initially placed on one peg with the largest disk on the bottom with increasingly smaller ones on top
- The goal is to move all of the disks from one peg to another under the following rules:
  - Move only one disk at a time
  - A larger disk cannot be put on top of a smaller one



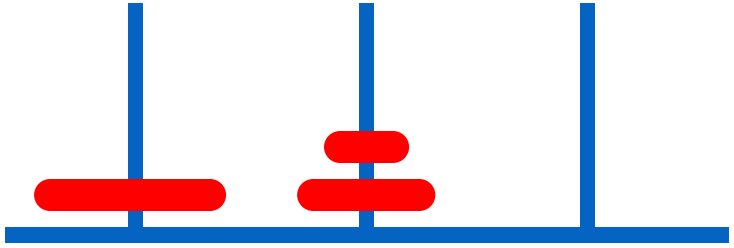
Towers of Hanoi

Original Configuration

Move 1

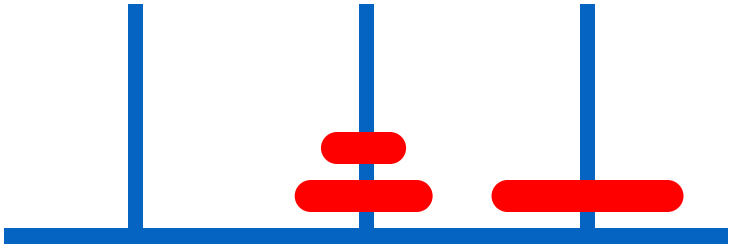


Move 2

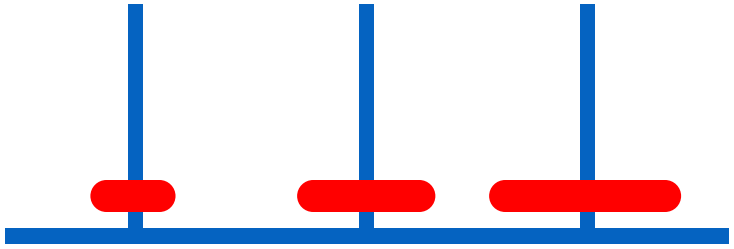


Move 3

Towers of Hanoi

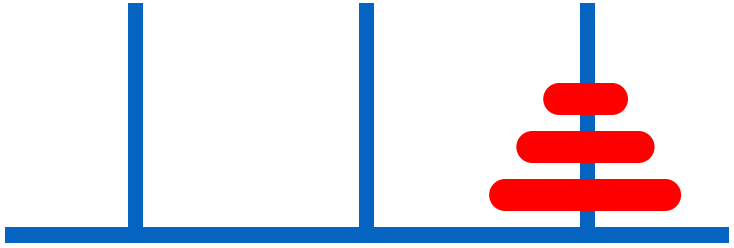


Move 4



Move 5

Move 6



Move 7 (done)

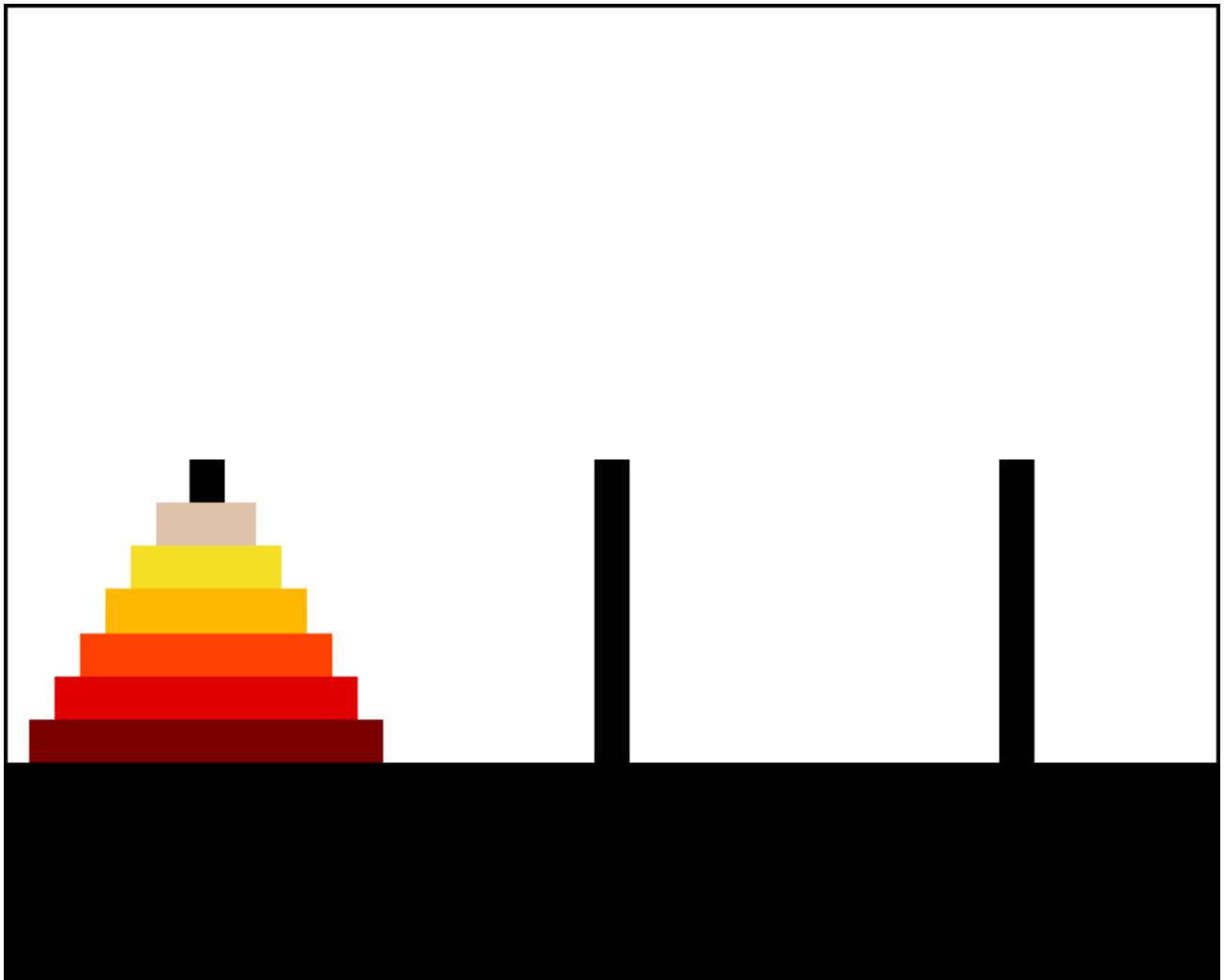
Towers of Hanoi

- An iterative solution to the Towers of Hanoi exists but more complex than the recursive one
- A recursive solution is much shorter and more elegant
- See SolveTowers.java
- See TowersOfHanoi.java

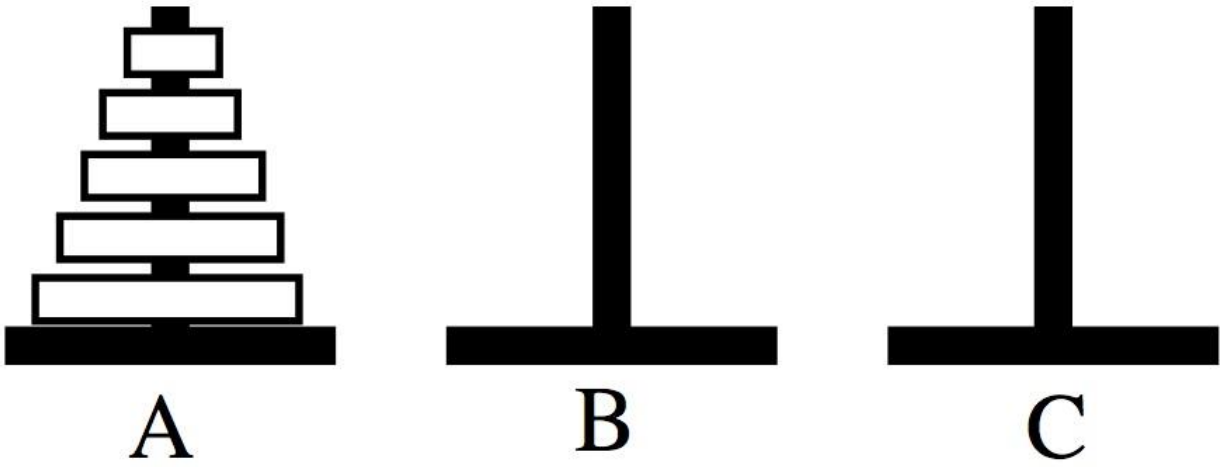
Iterative Solution

- For an even number of disks:
  - make the legal move between pegs A and B
  - make the legal move between pegs A and C
  - make the legal move between pegs B and C
  - repeat until complete
- For an odd number of disks:
  - make the legal move between pegs A and C
  - make the legal move between pegs A and B
  - make the legal move between pegs C and B
  - repeat until complete

Iterative Solution



Recursive Solution



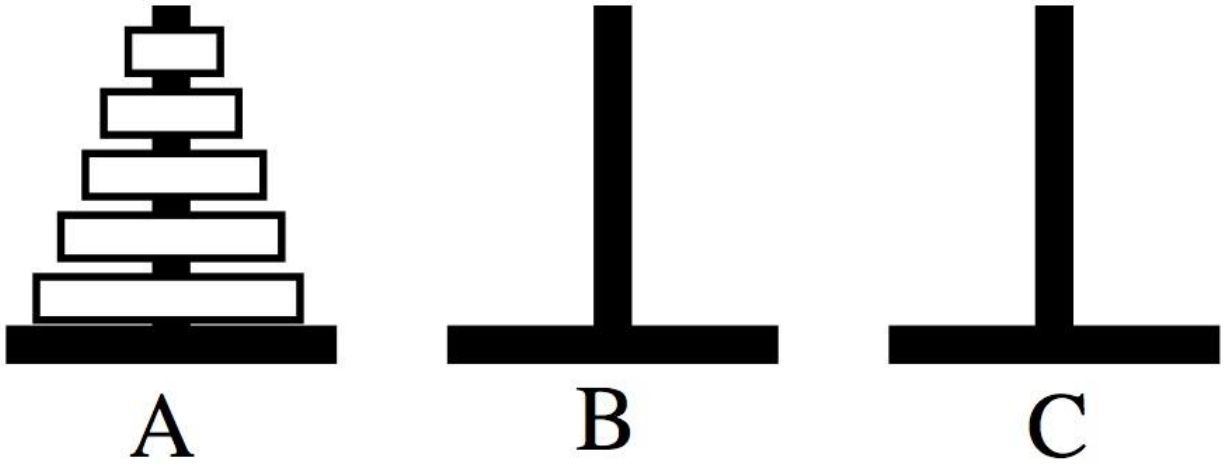
Recursive Solution

```
//-----------------------------------------------------------------
   //  Performs the initial call to moveTower to solve the puzzle.
   //  Moves the disks from tower 1 to tower 3 using tower 2.
   //-----------------------------------------------------------------
   public void solve ()
   {
      moveTower (totalDisks, 1, 3, 2);
   }



    private void moveTower (int numDisks, int start, int end, int temp)
       {




    }
```

Recursive Solution

- Break the problem down into a collection of smaller problems
- Assume you have n disks

1. Move n disks from A to C by moving a subtower of n-1 disks out of the way (to B)
2. Move one disk from A to C
3. Move the subtower to C
4. Base case of 1 disk



```java
//********************************************************************
//  SolveTowers.java       Author: Lewis/Loftus
//
//  Demonstrates recursion.
//********************************************************************

public class SolveTowers
{
   //-----------------------------------------------------------------
   //  Creates a TowersOfHanoi puzzle and solves it.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      TowersOfHanoi towers = new TowersOfHanoi (4);

      towers.solve();
   }
}
```

```
//*****************                                    ********************
//  SolveTowers.jav
//
//  Demonstrates re                                    ********************
//*****************

public class Solve
{
   //--------------                                    --------------------
   //  Creates a To                                    .
   //--------------                                    --------------------
   public static vo
   {
      TowersOfHano

      towers.solve
   }
}
```

Output

```
Move one disk from 1 to 2
Move one disk from 1 to 3
Move one disk from 2 to 3
Move one disk from 1 to 2
Move one disk from 3 to 1
Move one disk from 3 to 2
Move one disk from 1 to 2
Move one disk from 1 to 3
Move one disk from 2 to 3
Move one disk from 2 to 1
Move one disk from 3 to 1
Move one disk from 2 to 3
Move one disk from 1 to 2
Move one disk from 1 to 3
Move one disk from 2 to 3
```

```
//********************************************************************
//  TowersOfHanoi.java       Author: Lewis/Loftus
//
//  Represents the classic Towers of Hanoi puzzle.
//********************************************************************

public class TowersOfHanoi
{
   private int totalDisks;

   //-----------------------------------------------------------------
   //  Sets up the puzzle with the specified number of disks.
   //-----------------------------------------------------------------
   public TowersOfHanoi (int disks)
   {
      totalDisks = disks;
   }

   //-----------------------------------------------------------------
   //  Performs the initial call to moveTower to solve the puzzle.
   //  Moves the disks from tower 1 to tower 3 using tower 2.
   //-----------------------------------------------------------------
   public void solve ()
   {
      moveTower (totalDisks, 1, 3, 2);
   }
```

continued

continued

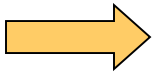
```
    //-----------------------------------------------------------------
    //  Moves the specified number of disks from one tower to another
    //  by moving a subtower of n-1 disks out of the way, moving one
    //  disk, then moving the subtower back. Base case of 1 disk.
    //-----------------------------------------------------------------
    private void moveTower (int numDisks, int start, int end, int temp)
    {
       if (numDisks == 1)
          moveOneDisk (start, end);
       else
       {
          moveTower (numDisks-1, start, temp, end);
          moveOneDisk (start, end);
          moveTower (numDisks-1, temp, end, start);
       }
    }

    //-----------------------------------------------------------------
    //  Prints instructions to move one disk from the specified start
    //  tower to the specified end tower.
    //-----------------------------------------------------------------
    private void moveOneDisk (int start, int end)
    {
       System.out.println ("Move one disk from " + start + " to " +
                           end);
    }
}
```

Outline

Recursive Thinking

Recursive Programming

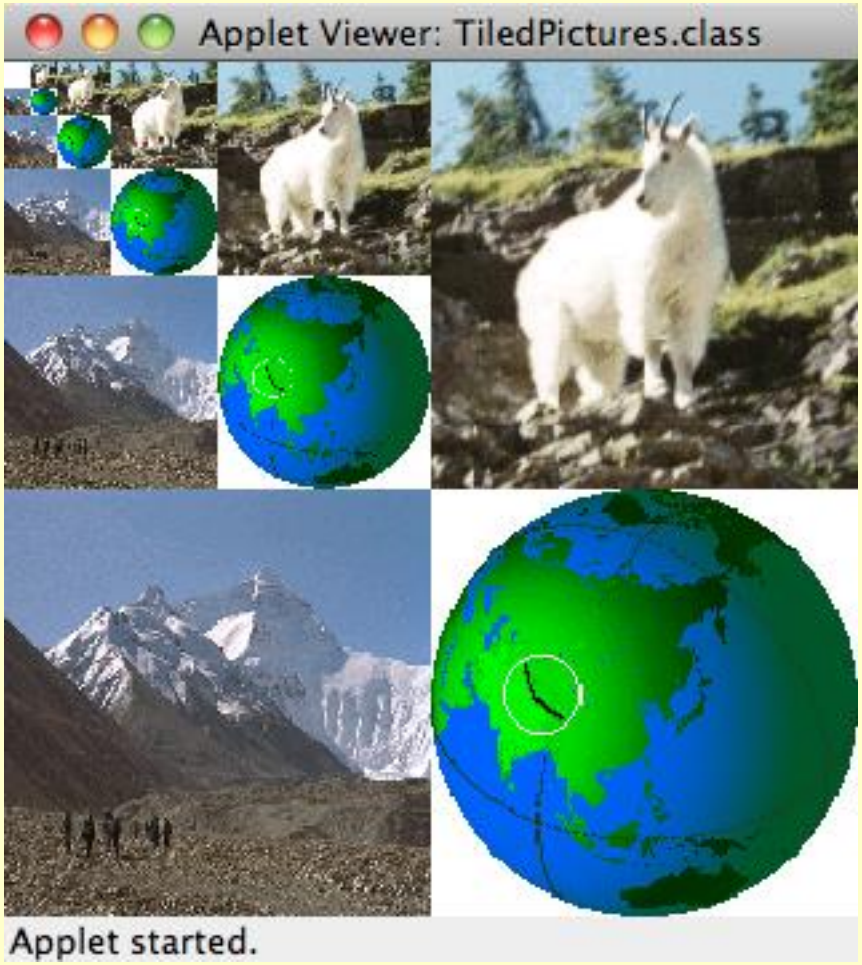
Using Recursion



Recursion in Graphics

Tiled Pictures

- Consider the task of repeatedly displaying a set of images in a mosaic
  - Three quadrants contain individual images
  - Upper-left quadrant repeats pattern
- The base case is reached when the area for the images shrinks to a certain size
- See TiledPictures.java


Applet Viewer: TiledPictures.class
Applet started.

```
//********************************************************************
//  TiledPictures.java       Author: Lewis/Loftus
//
//  Demonstrates the use of recursion.
//********************************************************************

import java.awt.*;
import javax.swing.JApplet;

public class TiledPictures extends JApplet
{
   private final int APPLET_WIDTH = 320;
   private final int APPLET_HEIGHT = 320;
   private final int MIN = 20;  // smallest picture size

   private Image world, everest, goat;
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Loads the images.
   //-----------------------------------------------------------------
   public void init()
   {
      world = getImage (getDocumentBase(), "world.gif");
      everest = getImage (getDocumentBase(), "everest.gif");
      goat = getImage (getDocumentBase(), "goat.gif");

      setSize (APPLET_WIDTH, APPLET_HEIGHT);
   }

   //-----------------------------------------------------------------
   //  Draws the three images, then calls itself recursively.
   //-----------------------------------------------------------------
   public void drawPictures (int size, Graphics page)
   {
      page.drawImage (everest, 0, size/2, size/2, size/2, this);
      page.drawImage (goat, size/2, 0, size/2, size/2, this);
      page.drawImage (world, size/2, size/2, size/2, size/2, this);

      if (size > MIN)
         drawPictures (size/2, page);
   }
```

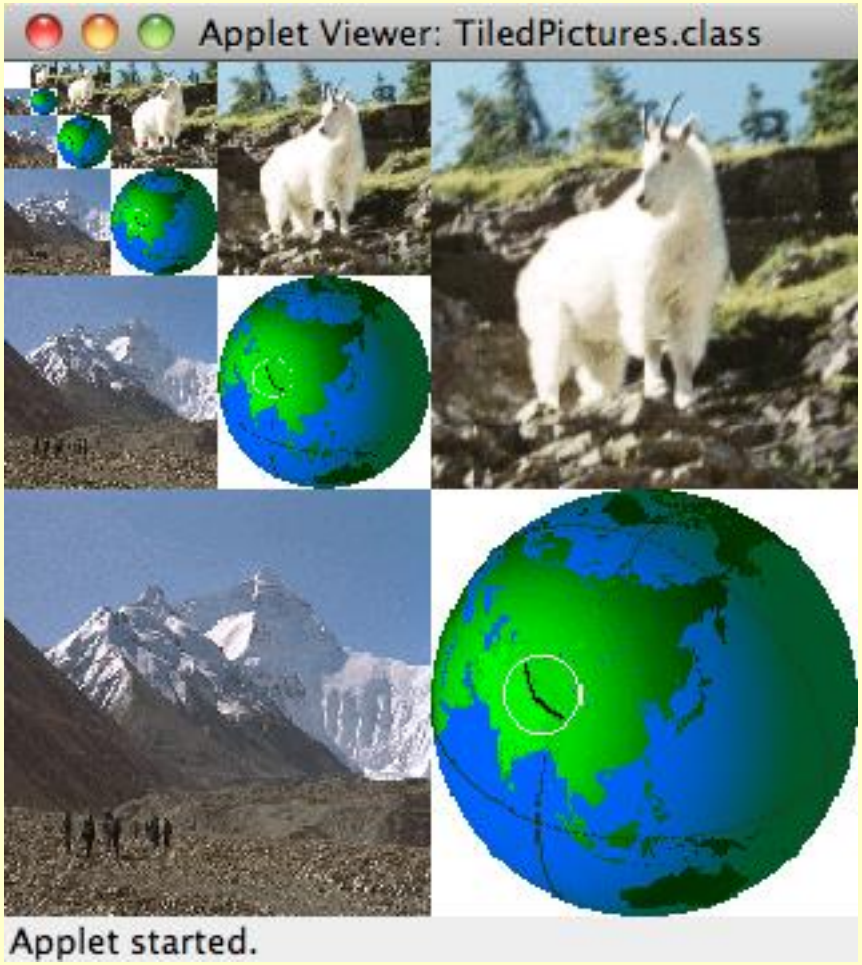
continue

continue

```
    //-----------------------------------------------------------------
    //  Performs the initial call to the drawPictures method.
    //-----------------------------------------------------------------
    public void paint (Graphics page)
    {
       drawPictures (APPLET_WIDTH, page);
    }
}
```

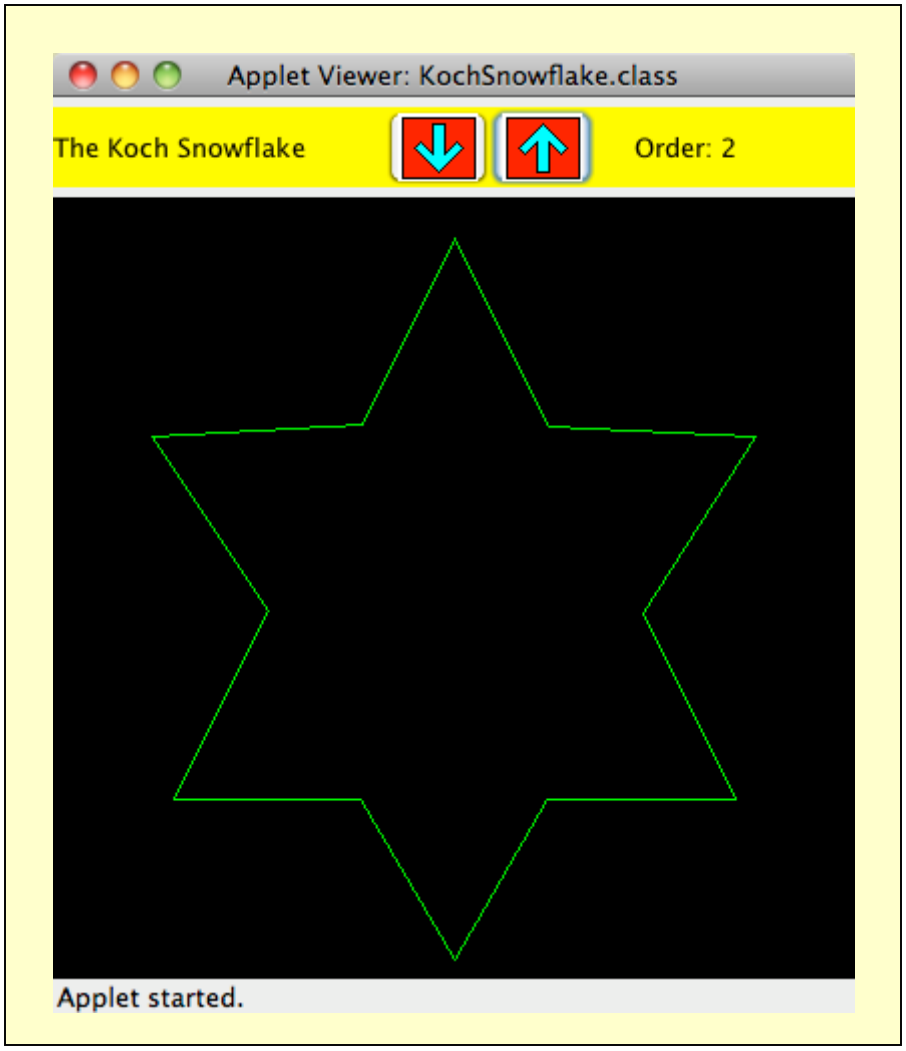
continue

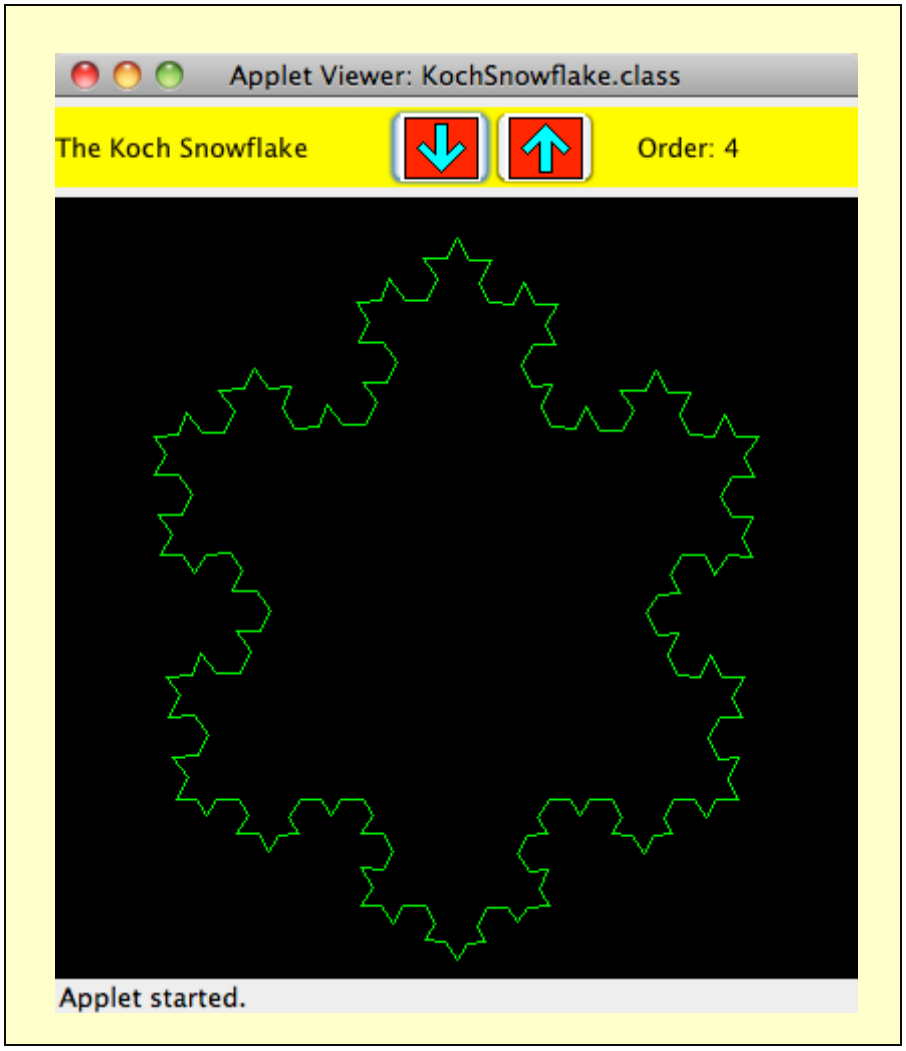
```
    //---------
    //  Perform
    //---------
    public void
    {
        drawPict
    }
}
```



Fractals

- A fractal is a geometric shape made up of the same pattern repeated in different sizes and orientations
- The Koch Snowflake is a particular fractal that begins with an equilateral triangle
- To get a higher order of the fractal, the sides of the triangle are replaced with angled line segments
- See KochSnowflake.java
- See KochPanel.java


Applet Viewer: KochSnowflake.class
The Koch Snowflake
Order: 2
Applet started.


Applet Viewer: KochSnowflake.class
The Koch Snowflake
Order: 4
Applet started.

```
//********************************************************************
//  KochSnowflake.java       Author: Lewis/Loftus
//
//  Demonstrates the use of recursion in graphics.
//********************************************************************

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class KochSnowflake extends JApplet implements ActionListener
{
   private final int APPLET_WIDTH = 400;
   private final int APPLET_HEIGHT = 440;

   private final int MIN = 1, MAX = 9;

   private JButton increase, decrease;
   private JLabel titleLabel, orderLabel;
   private KochPanel drawing;
   private JPanel appletPanel, tools;
```

continue

continue

```
//-----------------------------------------------------------------
//  Sets up the components for the applet.
//-----------------------------------------------------------------
public void init()
{
   tools = new JPanel ();
   tools.setLayout (new BoxLayout(tools, BoxLayout.X_AXIS));
   tools.setPreferredSize (new Dimension (APPLET_WIDTH, 40));
   tools.setBackground (Color.yellow);
   tools.setOpaque (true);

   titleLabel = new JLabel ("The Koch Snowflake");
   titleLabel.setForeground (Color.black);

   increase = new JButton (new ImageIcon ("increase.gif"));
   increase.setPressedIcon (new ImageIcon ("increasePressed.gif"));
   increase.setMargin (new Insets (0, 0, 0, 0));
   increase.addActionListener (this);

   decrease = new JButton (new ImageIcon ("decrease.gif"));
   decrease.setPressedIcon (new ImageIcon ("decreasePressed.gif"));
   decrease.setMargin (new Insets (0, 0, 0, 0));
   decrease.addActionListener (this);
```

continue

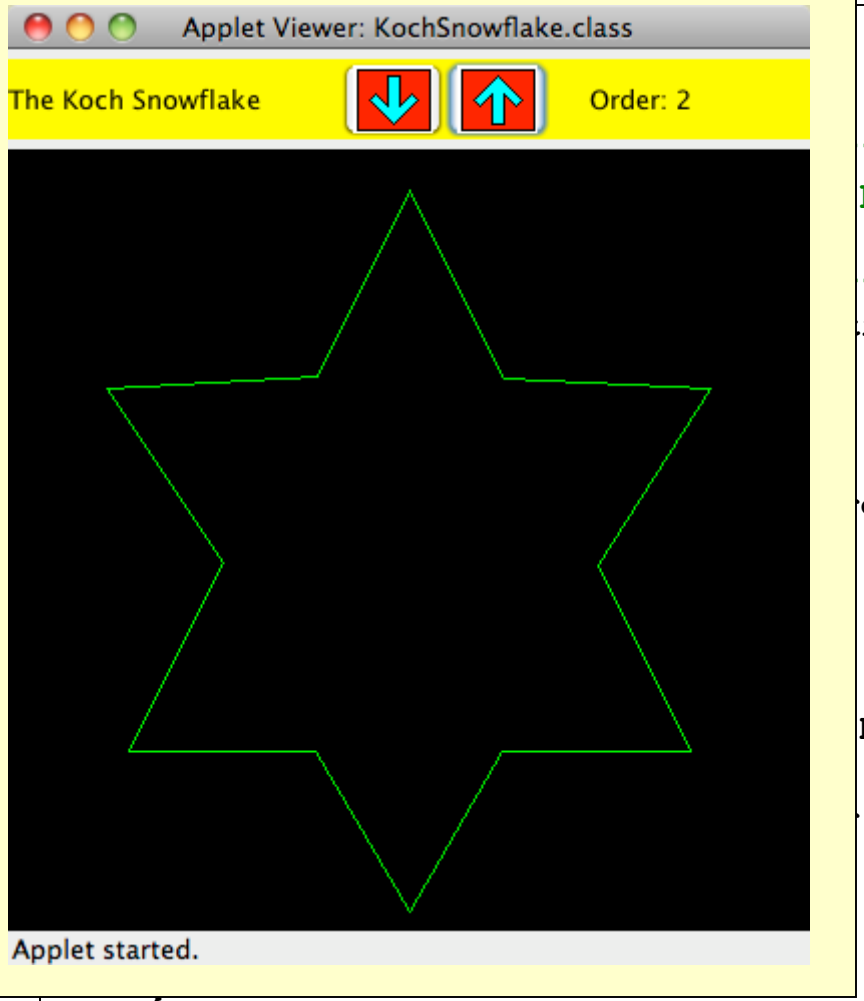
continue

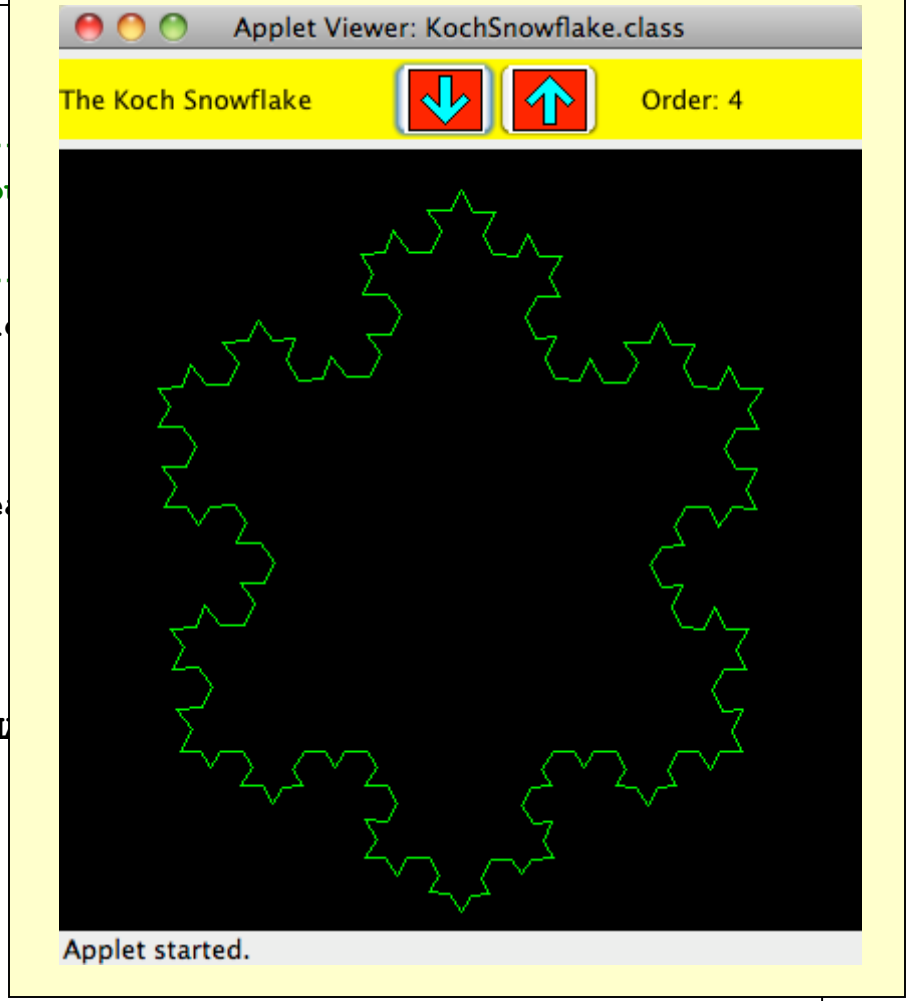
```
        orderLabel = new JLabel ("Order: 1");
        orderLabel.setForeground (Color.black);

        tools.add (titleLabel);
        tools.add (Box.createHorizontalStrut (40));
        tools.add (decrease);
        tools.add (increase);
        tools.add (Box.createHorizontalStrut (20));
        tools.add (orderLabel);

        drawing = new KochPanel (1);

        appletPanel = new JPanel();
        appletPanel.add (tools);
        appletPanel.add (drawing);

        getContentPane().add (appletPanel);

        setSize (APPLET_WIDTH, APPLET_HEIGHT);
    }
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Determines which button was pushed, and sets the new order
    //  if it is in range.
    //-----------------------------------------------------------------
    public void actionPerformed (ActionEvent event)
    {
       int order = drawing.getOrder();

       if (event.getSource() == increase)
          order++;
       else
          order--;

       if (order >= MIN && order <= MAX)
       {
          orderLabel.setText ("Order: " + order);
          drawing.setOrder (order);
          repaint();
       }
    }
}
```


Applet Viewer: KochSnowflake.class
The Koch Snowflake
Order: 2
Applet started.


Applet Viewer: KochSnowflake.class
The Koch Snowflake
Order: 4
Applet started.

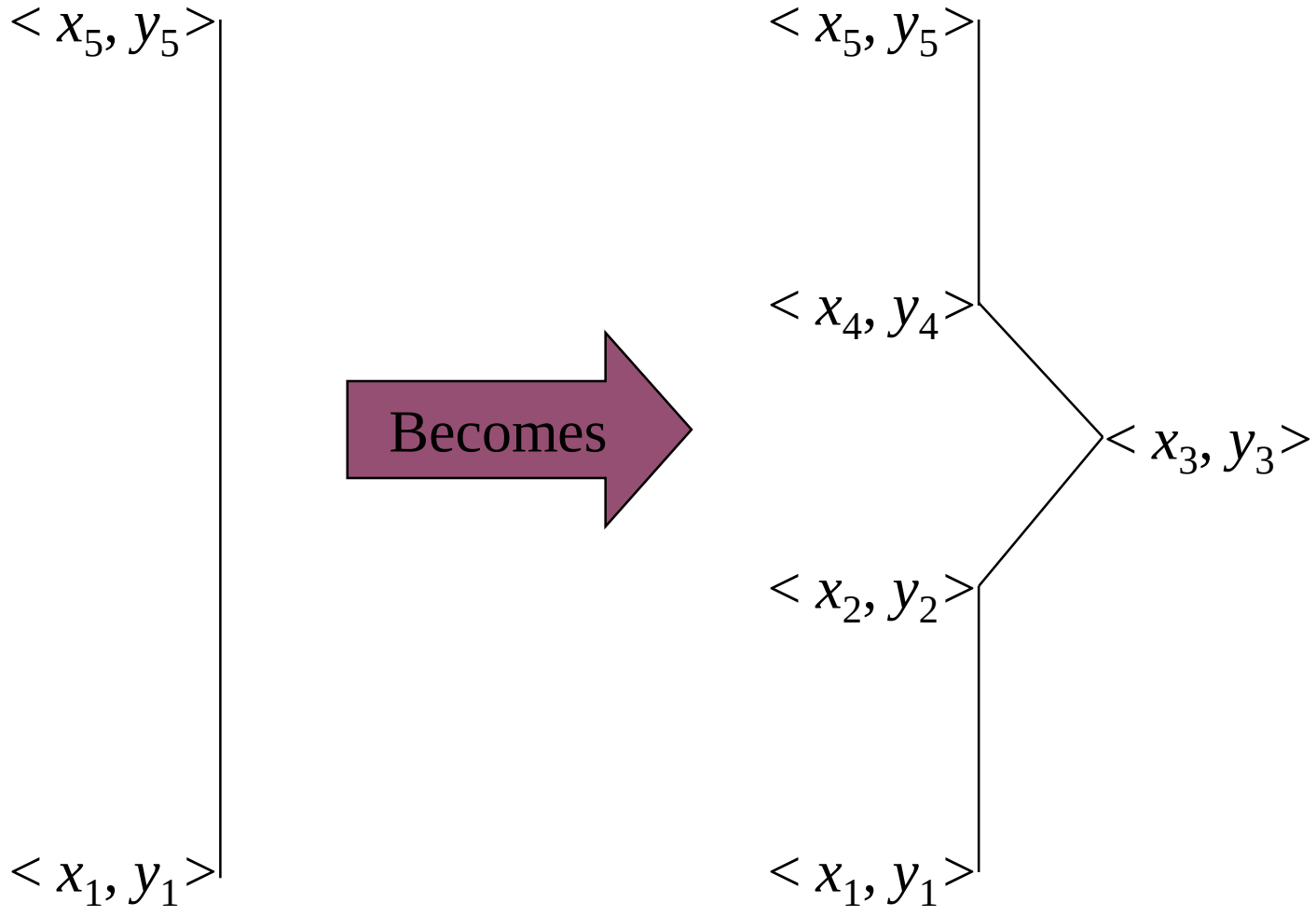
```
    }
}
```

Koch Snowflakes

```
public void drawFractal (int order, int x1, int y1, int x5, int
y5, Graphics page)
```

```
//-----------------------------------------------------------
----
   //  Performs the initial calls to the drawFractal method.
   //-----------------------------------------------------------
-------
   public void paintComponent (Graphics page)
   {
```

Koch Snowflakes



```
//********************************************************************
//  KochPanel.java       Author: Lewis/Loftus
//
//  Represents a drawing surface on which to paint a Koch Snowflake.
//********************************************************************

import java.awt.*;
import javax.swing.JPanel;

public class KochPanel extends JPanel
{
   private final int PANEL_WIDTH = 400;
   private final int PANEL_HEIGHT = 400;

   private final double SQ = Math.sqrt(3.0) / 6;

   private final int TOPX = 200, TOPY = 20;
   private final int LEFTX = 60, LEFTY = 300;
   private final int RIGHTX = 340, RIGHTY = 300;

   private int current;  // current order
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Draws the fractal recursively. The base case is order 1 for
   //  which a simple straight line is drawn. Otherwise three
   //  intermediate points are computed, and each line segment is
   //  drawn as a fractal.
   //-----------------------------------------------------------------
   public void drawFractal (int order, int x1, int y1, int x5, int y5,
                            Graphics page)
   {
      int deltaX, deltaY, x2, y2, x3, y3, x4, y4;

      if (order == 1)
         page.drawLine (x1, y1, x5, y5);
      else
      {
         deltaX = x5 - x1;  // distance between end points
         deltaY = y5 - y1;

         x2 = x1 + deltaX / 3;  // one third
         y2 = y1 + deltaY / 3;

         x3 = (int) ((x1+x5)/2 + SQ * (y1-y5));  // tip of projection
         y3 = (int) ((y1+y5)/2 + SQ * (x5-x1));
```

continue

continue

```
            x4 = x1 + deltaX * 2/3;  // two thirds
            y4 = y1 + deltaY * 2/3;

            drawFractal (order-1, x1, y1, x2, y2, page);
            drawFractal (order-1, x2, y2, x3, y3, page);
            drawFractal (order-1, x3, y3, x4, y4, page);
            drawFractal (order-1, x4, y4, x5, y5, page);
         }
      }

      //-----------------------------------------------------------------
      //  Performs the initial calls to the drawFractal method.
      //-----------------------------------------------------------------
      public void paintComponent (Graphics page)
      {
         super.paintComponent (page);

         page.setColor (Color.green);

         drawFractal (current, TOPX, TOPY, LEFTX, LEFTY, page);
         drawFractal (current, LEFTX, LEFTY, RIGHTX, RIGHTY, page);
         drawFractal (current, RIGHTX, RIGHTY, TOPX, TOPY, page);
      }
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Sets the fractal order to the value specified.
    //-----------------------------------------------------------------
    public void setOrder (int order)
    {
       current = order;
    }

    //-----------------------------------------------------------------
    //  Returns the current order.
    //-----------------------------------------------------------------
    public int getOrder ()
    {
       return current;
    }
}
```

More Examples

Calculate Power

- Write a power method that works recursively
- Compute n^m recursively
- Power.java

Calculate Power

```
public static int pow(int base, int exp)
{       int result = 0;
        if(exp == 0)
                result = 1;
        else
                result = base * pow(base, exp - 1);

        return result;
}
```

Reverse a String Recursively

- Get a String from the user and reverse it recursively

- ReverseString.java

Reverse a String Recursively

```
public static String reverseRecursively(String str) {

    //base case to handle one char string and empty string
    if (str.length() < 2) {
        return str;
    }
    return reverseRecursively(str.substring(1)) + str.charAt(0);
```

Fibonacci Series

- Each number in the series is sum of two previous numbers
  - e.g., 0, 1, 1, 2, 3, 5, 8, 13, 21…

    fibonacci(n)=?

  - Fibonacci.java

Fibonacci Series

- Each number in the series is sum of two previous numbers
  - e.g., 0, 1, 1, 2, 3, 5, 8, 13, 21…
  - Hint: fibonacci(n) = fibonacci(n - 1) + fibonacci(n – 2)
  - fibonacci(0) and fibonacci(1) are base cases
  - fibonacci(0) = 0
    fibonacci(1) = 1

- Fibonacci.java

section_3/RecursiveFib.java

```
 1  import java.util.Scanner;
 2
 3  /**
 4     This program computes Fibonacci numbers using a recursive method.
 5  */
 6  public class RecursiveFib
 7  {
 8     public static void main(String[] args)
 9     {
10        Scanner in = new Scanner(System.in);
11        System.out.print("Enter n: ");
12        int n = in.nextInt();
13
14        for (int i = 1; i <= n; i++)
15        {
16           long f = fib(i);
17           System.out.println("fib(" + i + ") = " + f);
18        }
19     }
20
```

Continued

section_3/RecursiveFib.java

```
21      /**
22         Computes a Fibonacci number.
23         @param n an integer
24         @return the nth Fibonacci number
25      */
26      public static long fib(int n)
27      {
28         if (n <= 2) { return 1; }
29         else { return fib(n - 1) + fib(n - 2); }
30      }
31   }
```

Program Run:

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

Anagrams

- Listing all the rearrangements of a word entered by the user:
  - East → program should list all 24 permutations, including eats, etas, teas, and non-words like tsae

Anagrams

- Listing all the rearrangements of a word entered by the user:
  - East → program should list all 24 permutations, including eats, etas, teas, and non-words like tsae
- Take each letter of the four-letter word
- Place that letter at the front of all the three-letter permutations of the remaining letters
- There will be four recursive calls to display all permutations of a four-letter word.
- Base case of our recursion would be when we reach a word with just one letter.

Anagrams

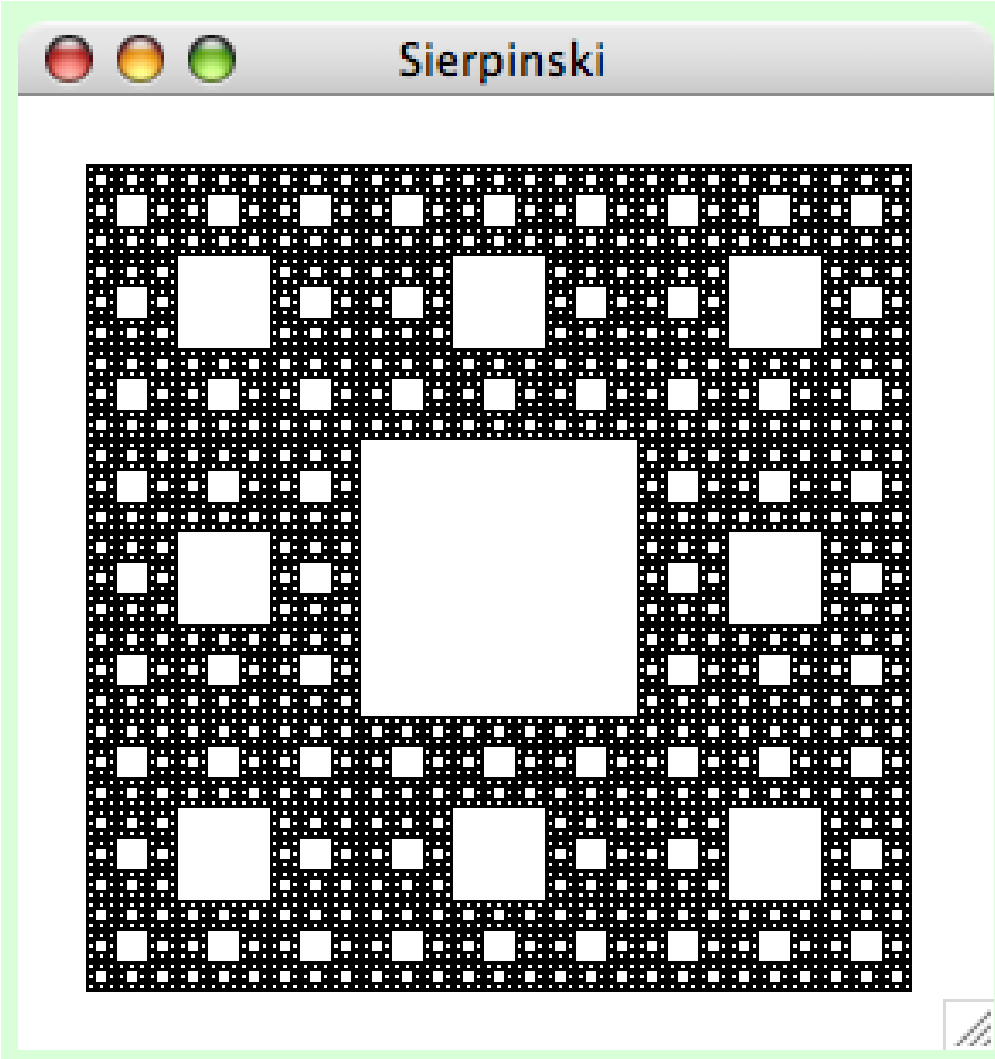
```
1  import acm.program.*;
2
3  public class Anagrams extends Program {
4      public void run() {
5          String word = readLine("Give a word to anagram: ");
6          printAnagrams("", word);
7      }
8
9      public void printAnagrams(String prefix, String word) {
10         if(word.length() <= 1) {
11             println(prefix + word);
12         } else {
13             for(int i = 0; i < word.length(); i++) {
14                 String cur = word.substring(i, i + 1);
15                 String before = word.substring(0, i); // letters before cur
16                 String after = word.substring(i + 1); // letters after cur
17                 printAnagrams(prefix + cur, before + after);
18             }
19         }
20     }
21 }
```

Sierpinski gasket

- Recursion can help in displaying complex patterns
- Pattern appears inside itself as a smaller version.
- Fractals are in fact a visual manifestation of the concept of recursion.

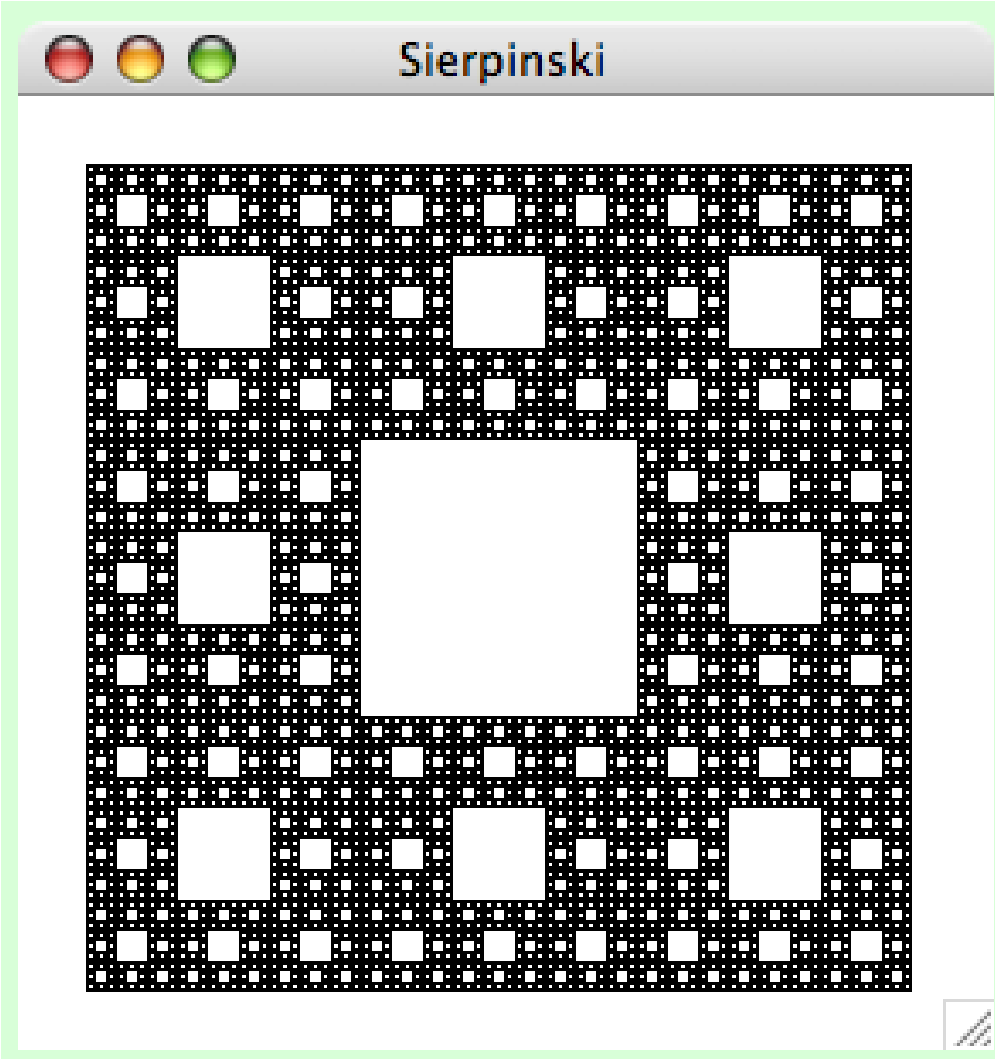
Sierpinski gasket

- Three parameters indicating the position of the gasket to be drawn;
  - x- and y-coordinates of the gasket's upper left corner
  - third will indicate how wide and tall the gasket should be.
- Base case will be when the side length goes below 3 pixels.



Sierpinski gasket

- Immediately draw a white box centered within the gasket, whose side length is 1/3 of the overall gasket's side length
- And then it will draw the eight smaller gaskets surrounding that box, each of whose side lengths is also 1/3 of the overall gasket's side length.
- Base case will be when the side length goes below 3 pixels.



Sierpinski gasket

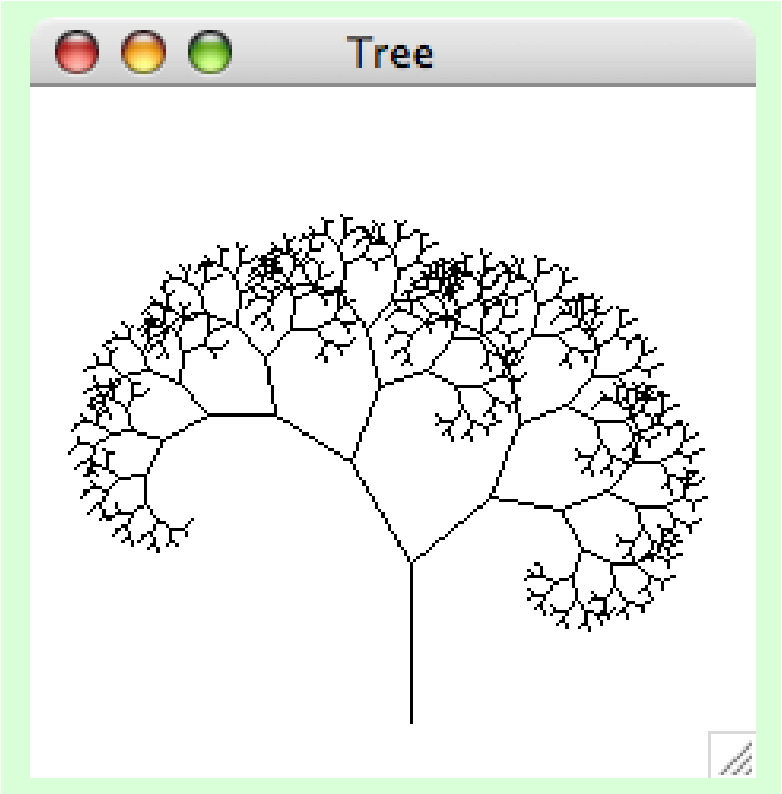
```
1  import java.awt.*;
2  import acm.program.*;
3  import acm.graphics.*;
4
5  public class Sierpinski extends GraphicsProgram {
6      public void run() {
7          // draw black background square
8          GRect box = new GRect(20, 20, 242, 242);
9          box.setFilled(true);
10         add(box);
11
12         // recursively draw all the white squares on top
13         drawGasket(20, 20, 243);
14     }
15
```

Sierpinski gasket

```
16      public void drawGasket(int x, int y, int side) {
17          // draw single white square in middle
18          int sub = side / 3; // length of sub-squares
19          GRect box = new GRect(x + sub, y + sub, sub - 1, sub - 1)
20          box.setFilled(true);
21          box.setColor(Color.WHITE);
22          add(box);
23
24          if(sub >= 3) {
25              // now draw eight sub-gaskets around the white square
26              drawGasket(x,           y,     sub);
27              drawGasket(x + sub,     y,     sub);
28              drawGasket(x + 2 * sub, y,     sub);
29              drawGasket(x,           y + sub, sub);
30              drawGasket(x + 2 * sub, y + sub, sub);
31              drawGasket(x,           y + 2 * sub, sub);
32              drawGasket(x + sub,     y + 2 * sub, sub);
33              drawGasket(x + 2 * sub, y + 2 * sub, sub);
34          }
35      }
36  }
```

Tree

- Tree consists of a trunk and two branches
- Each branch appears exactly the same as the overall tree
  - but smaller (75% left branch and 66% right branch)
  - rotated a bit (30° counterclockwise for the left branch, 50° clockwise for the right)
- Four parameters to indicate the trunk of the tree to be drawn
  - x and y coordinates of the trunk's base
  - length of the trunk
  - the angle of the trunk
- Base case will be when the length is at most 2 pixels



Tree

```
1  import java.awt.*;
2  import acm.program.*;
3  import acm.graphics.*;
4
5  public class Tree extends GraphicsProgram {
6      public void run() {
7          drawTree(120, 200, 50, 90);
8      }
9
10     public void drawTree(double x0, double y0, double len, double angle) {
11         if(len > 2) {
12             double x1 = x0 + len * GMath.cosDegrees(angle);
13             double y1 = y0 - len * GMath.sinDegrees(angle);
14
15             add(new GLine(x0, y0, x1, y1));
16             drawTree(x1, y1, len * 0.75, angle + 30);
17             drawTree(x1, y1, len * 0.66, angle - 50);
18         }
19     }
20 }
```

Greatest Common Divisor

- We can efficiently compute the greatest common divisor using Euclid's algorithm:
- Write the recursive and non-recursive versions
- Euclid.java

Greatest Common Divisor

- We can efficiently compute the greatest common divisor using Euclid's algorithm:
- Hint: for positive integers p and q:
  - If p > q, the gcd of p and q is the same as the gcd of q and (p % q)

- Write the recursive and non-recursive versions

- Euclid.java

Greatest Common Divisor

```
public class Euclid {

   // recursive implementation
   public static int gcd(int p, int q) {

      if (q == 0) return p;
      else return gcd(q, p % q);
   }

   // non-recursive implementation
   public static int gcd2(int p, int q) {

      while (q != 0) {
         int temp = q;
         q = p % q;
         p = temp;
      }
      return p;
   }
```

Integer to Binary Conversion

- Take an integer and printout the binary representation

- IntegerToBinary.java

Integer to Binary Conversion

- Take an integer and printout the binary representation
- Hint: repeatedly divide N into 2 and read the remainders backwards

- IntegerToBinary.java

Integer to Binary Conversion

```
public static void convert(int n) {
    if (n == 0)
        return;

    convert(n / 2);
    System.out.print(n % 2);
  }
```

Sorted Array

- Suppose you have an array of integers.
- Write a recursive Java method that returns true if the array is sorted from smallest to largest, and false otherwise.

```
public static boolean isSorted(int[] a)
 {
   boolean result = false;
   if(a.length < 2)
   {
     result = true;
   }
   else
   {
     if(a[0] <= a[1])
     {
       result = Sorted.isSorted(Arrays.copyOfRange(a, 1, a.length));
     }
   }
   return result;
 }
```

Search

- Design a method that returns true if element n is a member of array x[] and false if not

Search

- Iterative solution:

```
public boolean search(int[] x, int n) {
        for(int i = 0; i < x.length, i++) {
                if (x[i] == n]) return true;
        }
        return false;
}
```

Search

- Recursive solution:

```
boolean search(int[] x, int size, int n) {
        if (size > 0) {
                if (x[size-1] == n) {
                        return true;
                } else {
                        return search(x, size-1, n);
                }
        }
        return false;
}
```

Faster Search

Faster Search

```
search(phonebook, name) {
        if only one page, scan for the name
        else
                open to the middle
                determine if name is before or after this page
                if before
                        search (first half of phonebook, name)
                else
                        search (second half of phonebook, name)
```

Faster Search

```
boolean binarySearch(int[] x, int start, int end, int n) {
        if (end < start) return false;
        int mid = (start+end) / 2;
        if (x[mid] == n) {
                return true;
        } else {
        if (x[mid] < n) {
                return search(x, mid+1, end, n);
        } else {
                return search(x, start, mid-1, n);
                }
        }
}
```