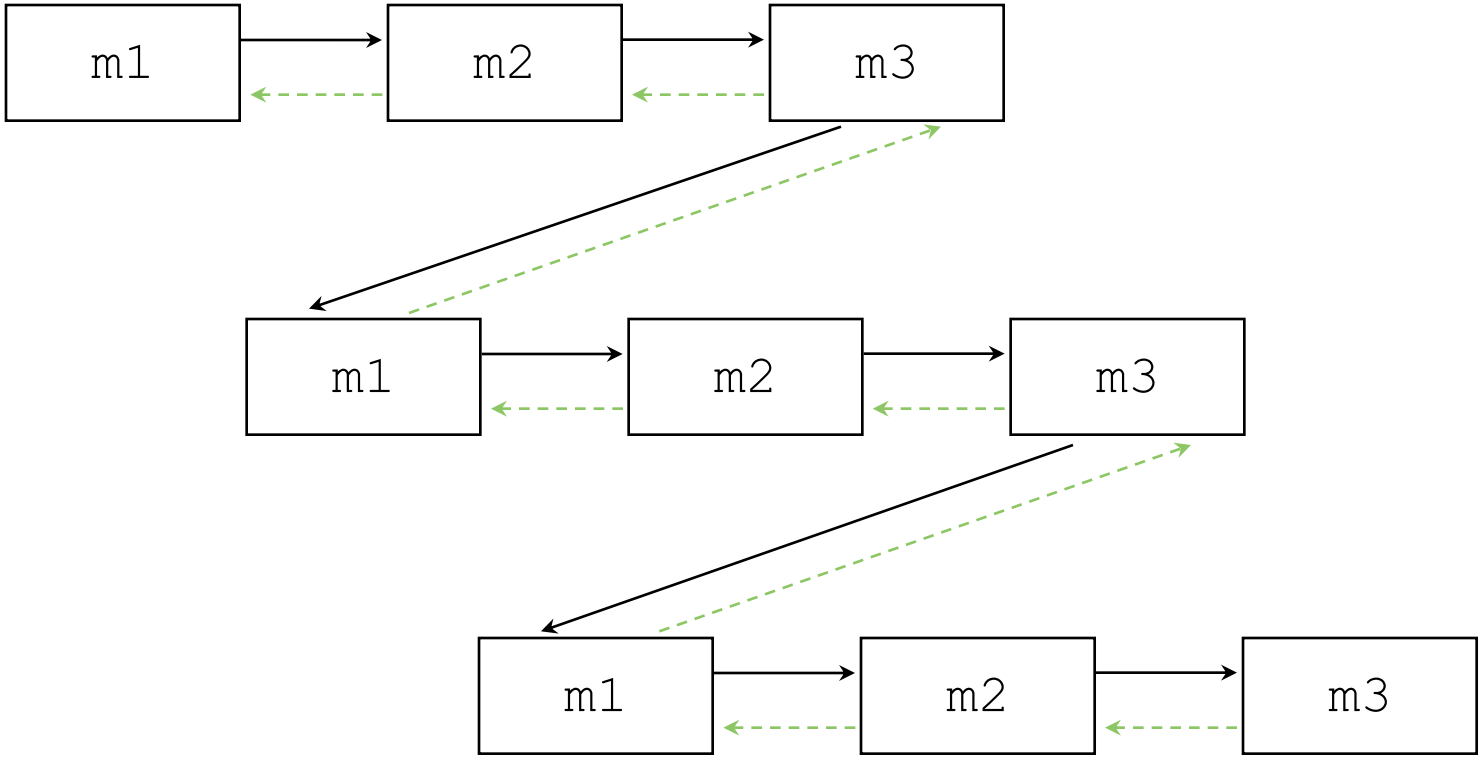
CHAPTER 13

RECURSION

Slides by Rick Giles

More Examples

Indirect Recursion



Maze Traversal

- We can use recursion to find a path through a maze
- From each location, we can search in each direction
- Recursion keeps track of the path through the maze
- The base case is an invalid move or reaching the final destination
- See `MazeSearch.java` (page 583)
- See `Maze.java` (page 584)

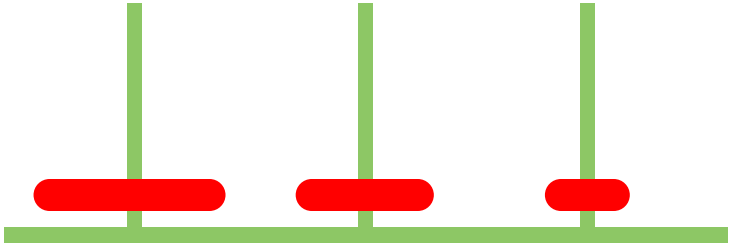
Towers of Hanoi

- The *Towers of Hanoi* is a puzzle made up of three vertical pegs and several disks that slide on 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:
  - We can move only one disk at a time
  - We cannot move a larger disk on top of a smaller one

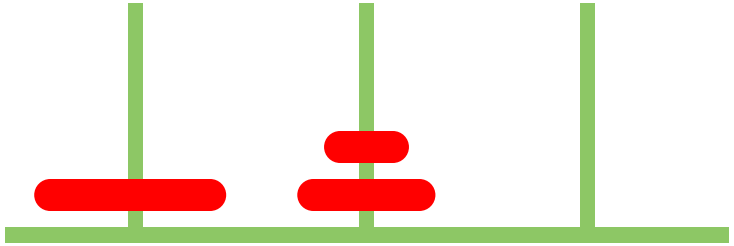
Towers of Hanoi

Original Configuration

Move 1

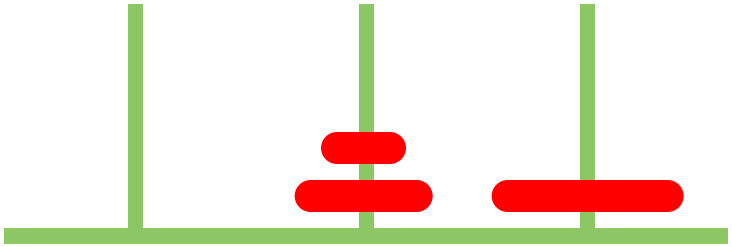


Move 2

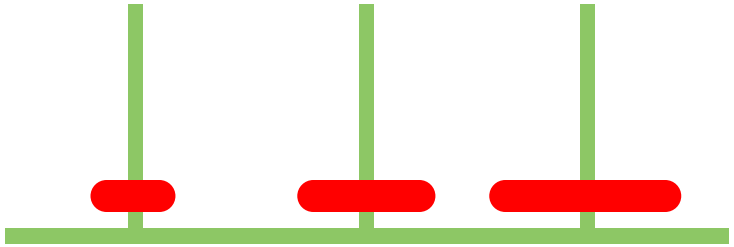


Move 3

Towers of Hanoi

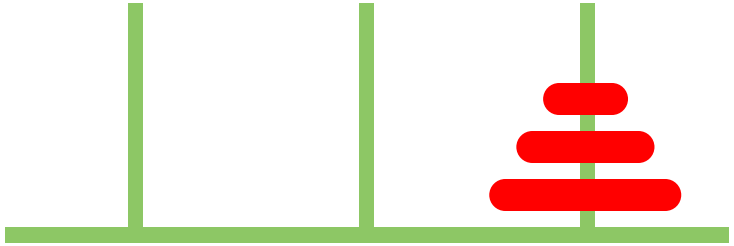


Move 4



Move 5

Move 6



Move 7 (done)

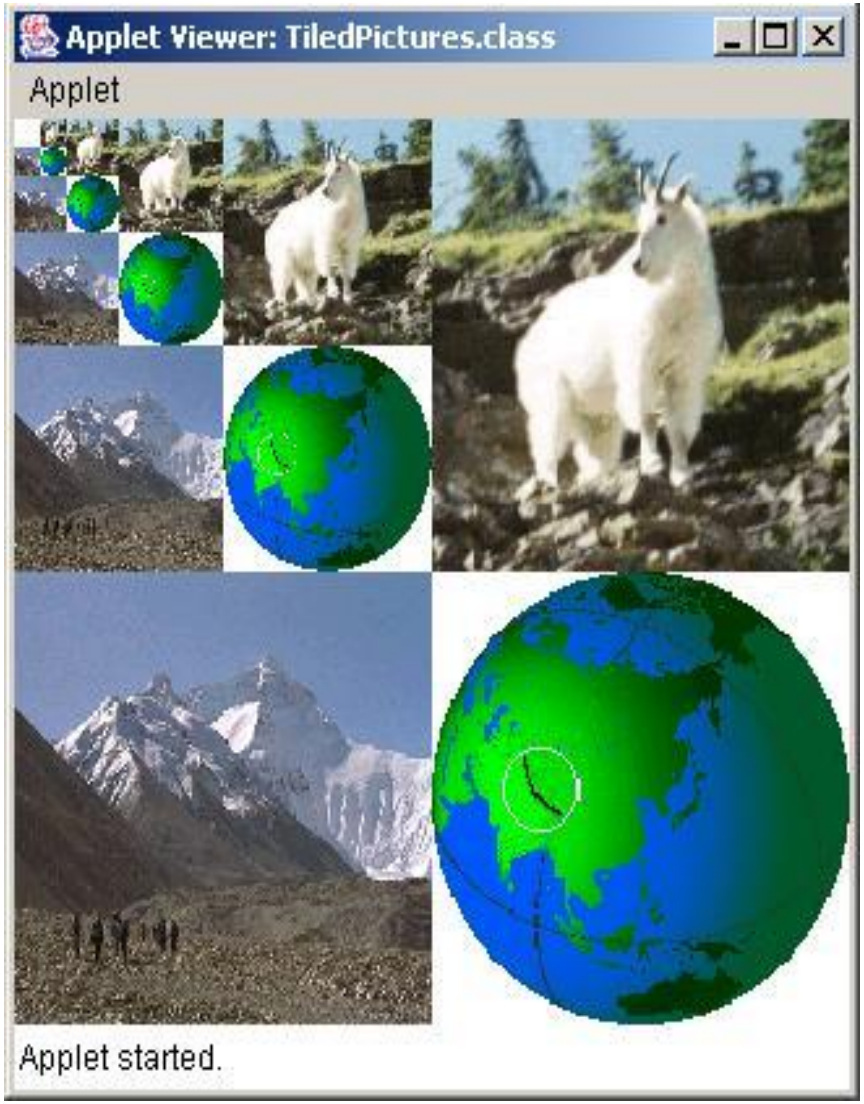
Towers of Hanoi

- An iterative solution to the Towers of Hanoi is quite complex
- A recursive solution is much shorter and more elegant
- See `SolveTowers.java` (page 590)
- See `TowersOfHanoi.java` (page 591)

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` (page 594)

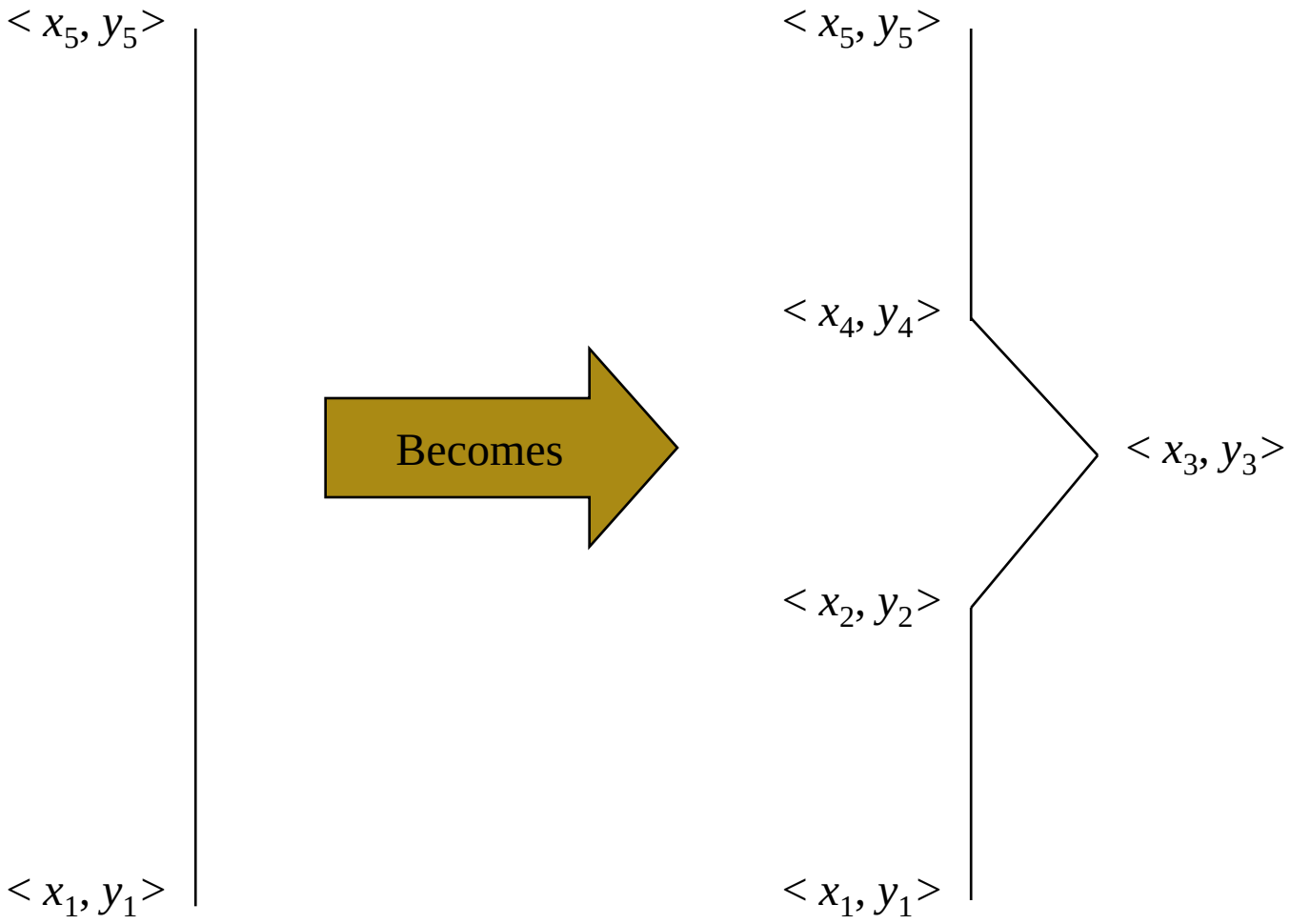
Tiled Pictures



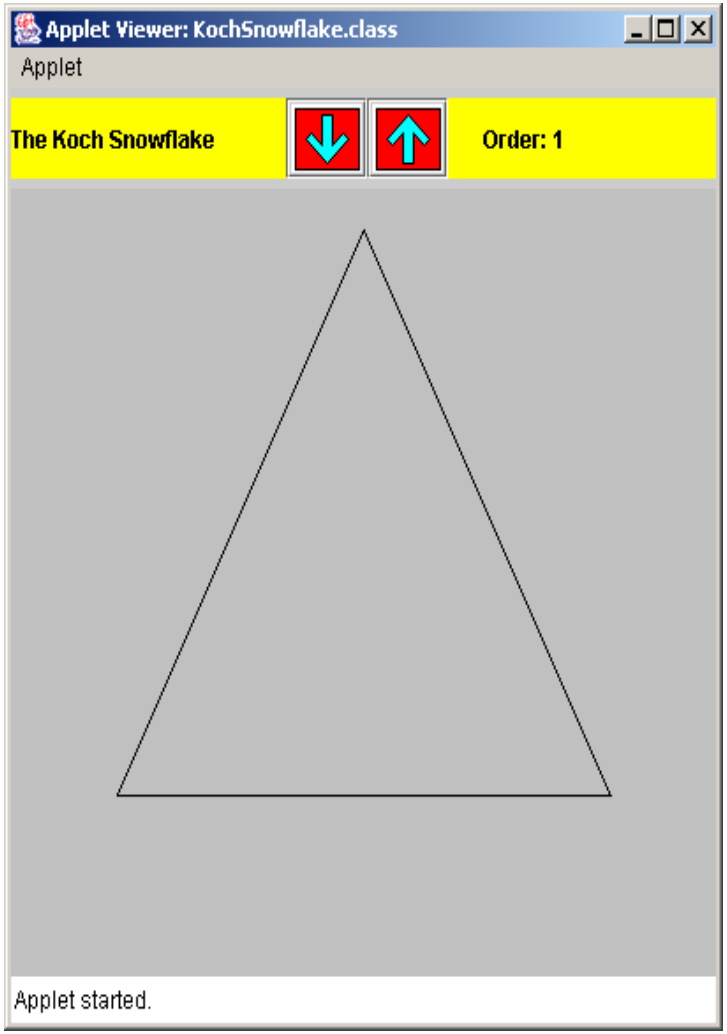
Fractals

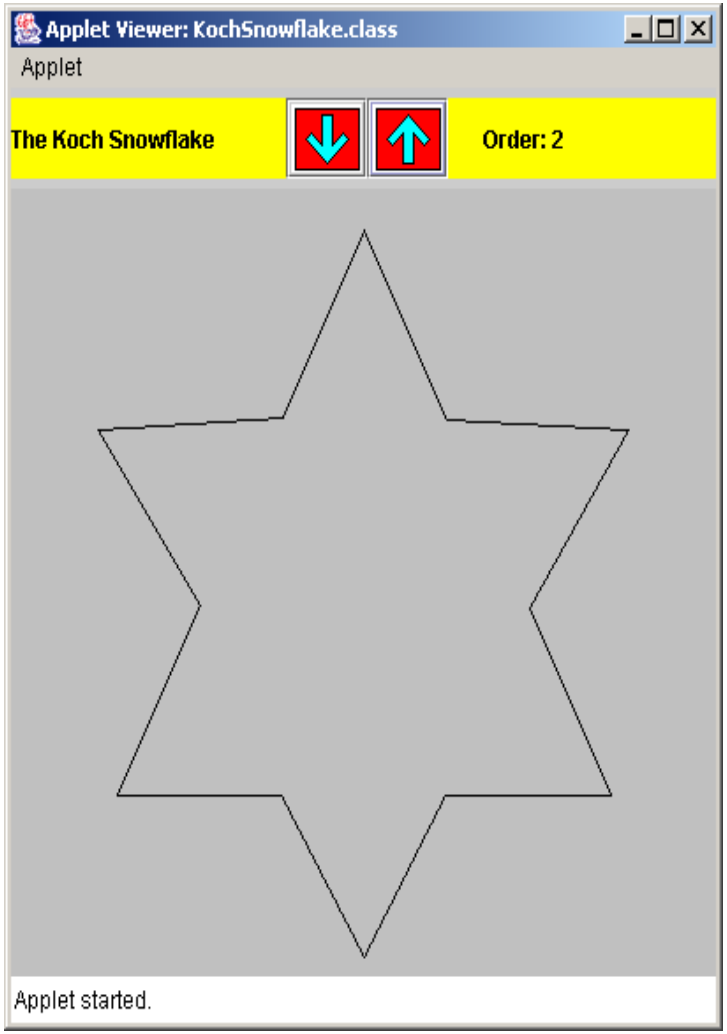
- A *fractal* is a geometric shape made up of the same pattern repeated in different sizes and orientations
- The *Koch Snowflake* is a particular fractal that begins with an equilateral triangle
- To get a higher order of the fractal, the sides of the triangle are replaced with angled line segments
- See `KochSnowflake.java` (page 597)
- See `KochPanel.java` (page 600)

Koch Snowflakes

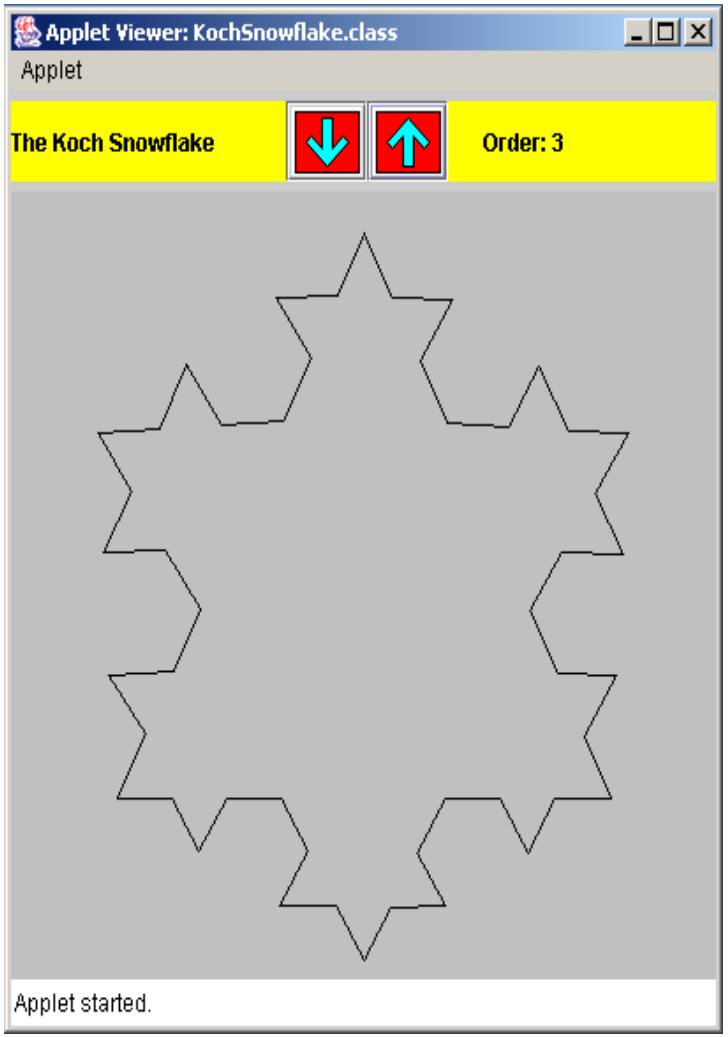


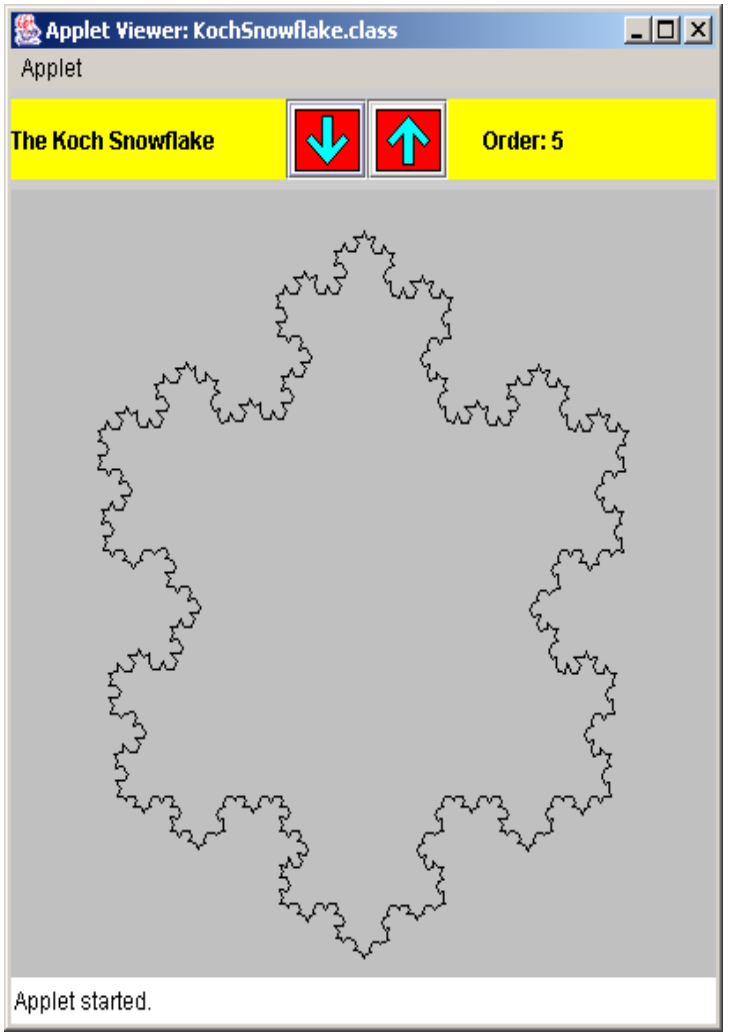
Koch Snowflakes





Koch Snowflakes





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

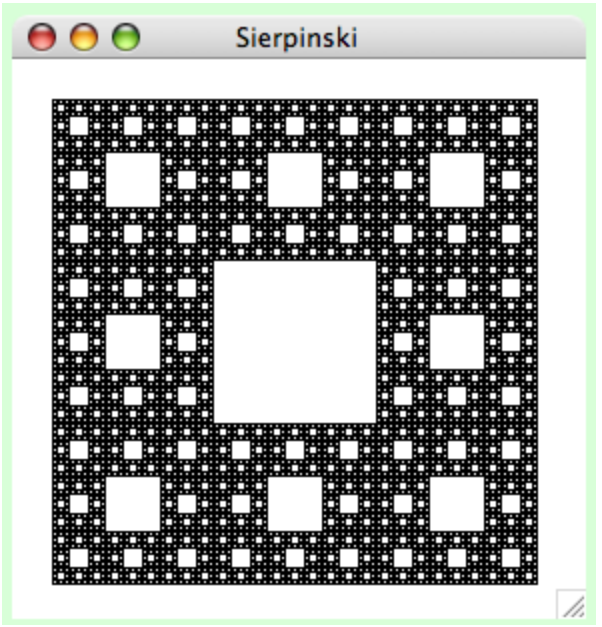
```
1  import acm.program.*;
2
3  public class Anagrams extends Program {
4      public void run() {
5          String word = readLine("Give a word to anagram: ");
6          printAnagrams("", word);
7      }
8
9      public void printAnagrams(String prefix, String word) {
10         if(word.length() <= 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.
- Composed of eight smaller Sierpinski gaskets arranged around the central white square

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.
- 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

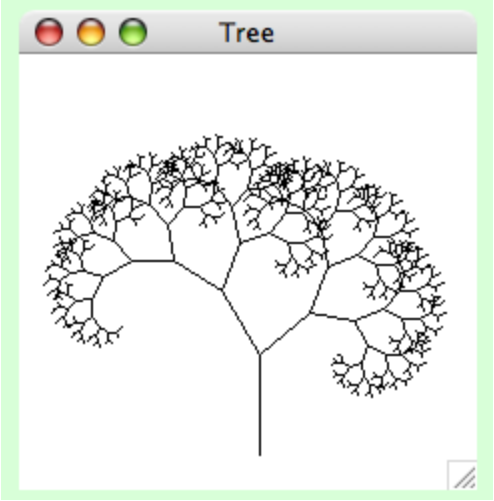
```
1  import java.awt.*;
2  import acm.program.*;
3  import acm.graphics.*;
4
5  public class Sierpinski extends GraphicsProgram {
6      public void run() {
7          // draw black background square
8          GRect box = new GRect(20, 20, 242, 242);
9          box.setFilled(true);
10         add(box);
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 }
```

Extra Slides

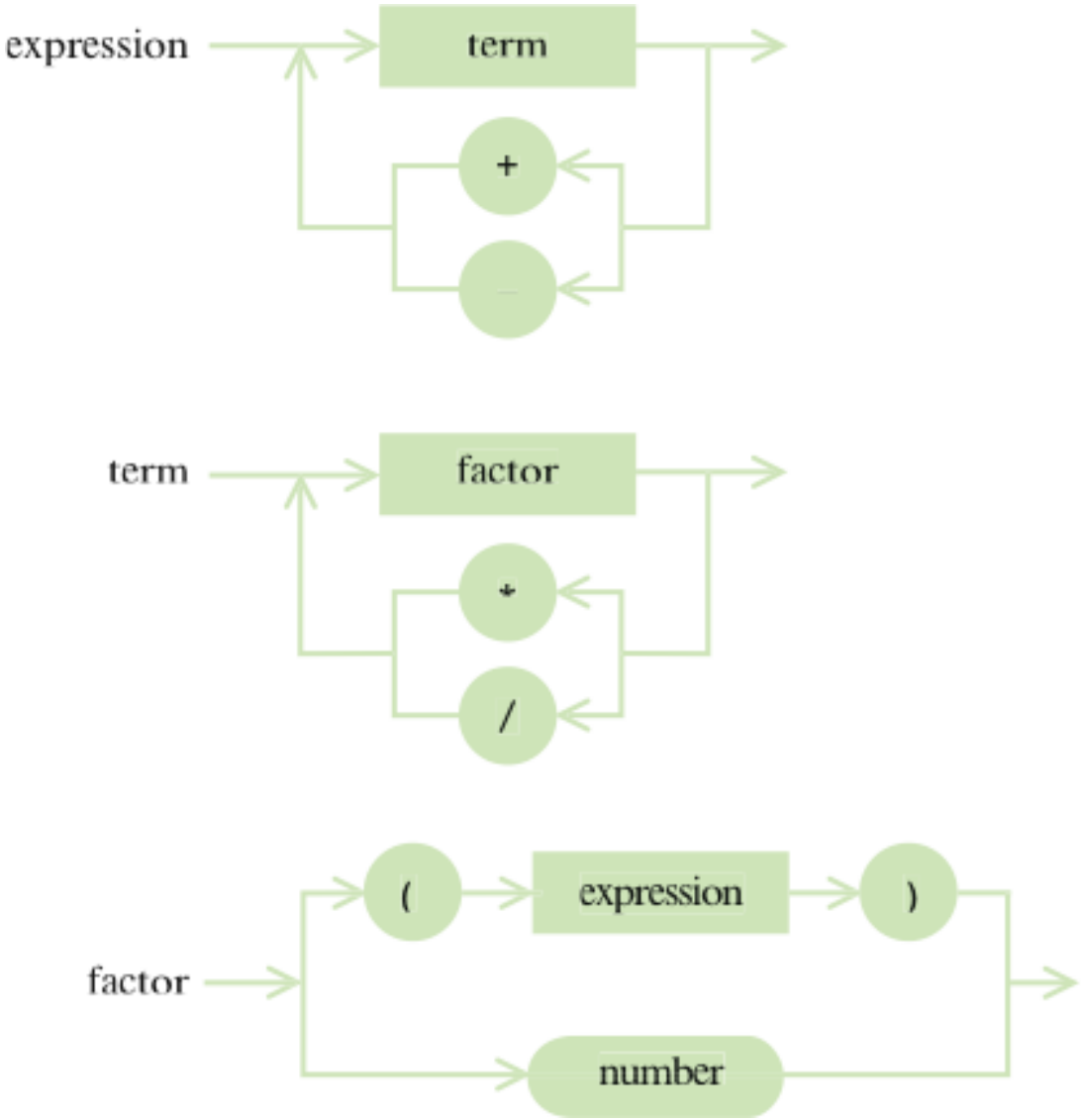
13.6 Mutual Recursion

- **Problem:** Compute the value of arithmetic expressions such as

```
3 + 4 * 5
(3 + 4) * 5
1 - (2 - (3 - (4 - 5)))
```

- Computing expression is complicated
  - `*` and `/` bind more strongly than `+` and `–`
  - Parentheses can be used to group subexpressions

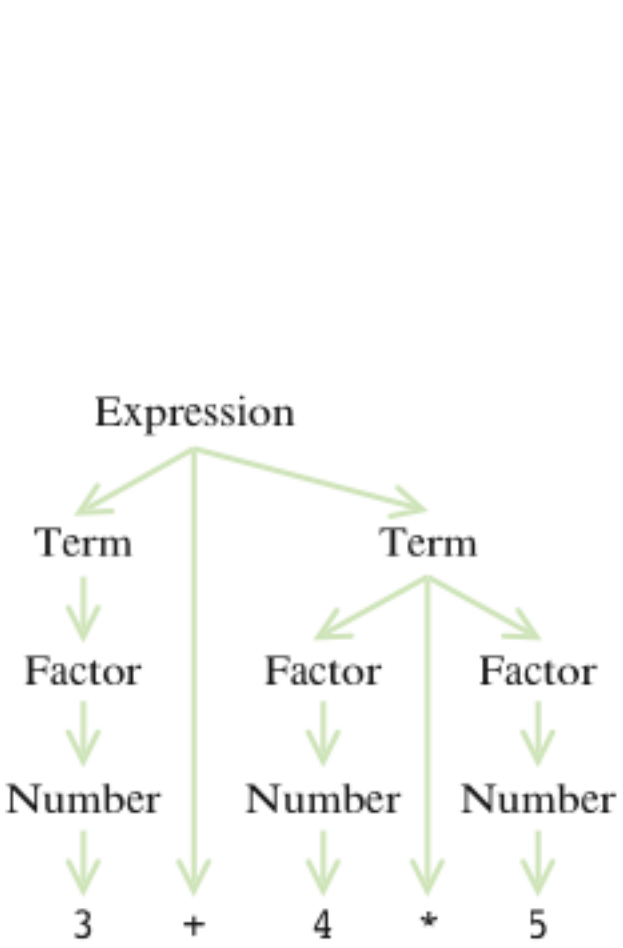
Syntax Diagrams for Evaluating an Expression

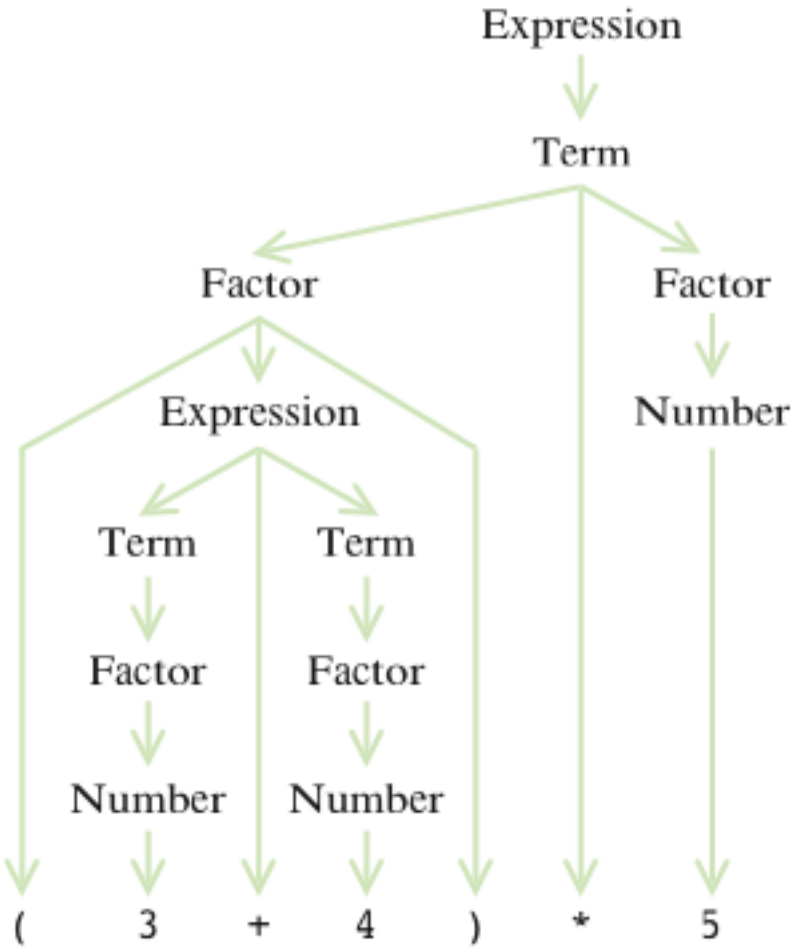


Mutual Recursion

- An *expression* can be broken down into a sequence of terms, separated by + or –
- Each *term* is broken down into a sequence of factors, separated by * or /
- Each *factor* is either a parenthesized expression or a number
- The syntax trees represent which operations should be carried out first

Syntax Trees for Two Expressions





Mutual Recursion

- In a mutual recursion, a set of cooperating methods calls each other repeatedly
- To compute the value of an expression, implement 3 methods that call each other recursively:
  - `getExpressionValue`
  - `getTermValue`
  - `getFactorValue`

getExpressionValue Method

```
public int getExpressionValue()
{
   int value = getTermValue();
   boolean done = false;
   while (!done)
   {
      String next = tokenizer.peekToken();
      if ("+".equals(next) || "-".equals(next))
      {
         tokenizer.nextToken(); // Discard "+" or "-"
         int value2 = getTermValue();
         if ("+".equals(next)) value = value + value2;
         else value = value - value2;
      }
      else done = true;
   }
   return value;
}
```

`getTermValue` Method

- The `getTermValue` method calls `getFactorValue` in the same way, multiplying or dividing the factor values

getFactorValue Method

```
public int getFactorValue()
{
   int value;
   String next =
   tokenpublic int getFactorValue()
{
   int value;
   String next = tokenizer.peekToken();
   if ("(".equals(next))
   {
      tokenizer.nextToken(); // Discard "("
      value = getExpressionValue();
      tokenizer.nextToken(); // Discard ")"
   }
   else
      value = Integer.parseInt(tokenizer.nextToken());
   return value;
}
```

Trace `(3 + 4) * 5`

To see the mutual recursion clearly, trace through the expression `(3+4)*5`:

- `getExpressionValue` calls `getTermValue`
  - `getTermValue` calls `getFactorValue`
    - `getFactorValue` consumes the `(` input
    - `getFactorValue` calls `getExpressionValue`
      - `getExpressionValue` returns eventually with the value of `7`, having consumed `3 + 4`. This is the recursive call.
    - `getFactorValue` consumes the `)` input
    - `getFactorValue` returns `7`
  - `getTermValue` consumes the inputs `*` and `5` and returns `35`
- `getExpressionValue` returns `35`

Evaluator.java

```
1  /**
2     A class that can compute the value of an arithmetic expression.
3  */
4  public class Evaluator
5  {
6     private ExpressionTokenizer tokenizer;
7
8     /**
9        Constructs an evaluator.
10       @param anExpression a string containing the expression
11       to be evaluated
12    */
13    public Evaluator(String anExpression)
14    {
15       tokenizer = new ExpressionTokenizer(anExpression);
16    }
17
```

Continued

Evaluator.java (cont.)

```
18      /**
19         Evaluates the expression.
20         @return the value of the expression.
21      */
22      public int getExpressionValue()
23      {
24         int value = getTermValue();
25         boolean done = false;
26         while (!done)
27         {
28            String next = tokenizer.peekToken();
29            if ("+".equals(next) || "-".equals(next))
30            {
31               tokenizer.nextToken(); // Discard "+" or "-"
32               int value2 = getTermValue();
33               if ("+".equals(next)) { value = value + value2; }
34               else { value = value - value2; }
35            }
36            else
37            {
38               done = true;
39            }
40         }
41         return value;
42      }
43
```

Continued

Evaluator.java (cont.)

```
44      /**
45         Evaluates the next term found in the expression.
46         @return the value of the term
47      */
48      public int getTermValue()
49      {
50         int value = getFactorValue();
51         boolean done = false;
52         while (!done)
53         {
54            String next = tokenizer.peekToken();
55            if ("*".equals(next) || "/".equals(next))
56            {
57               tokenizer.nextToken();
58               int value2 = getFactorValue();
59               if ("*".equals(next)) { value = value * value2; }
60               else { value = value / value2; }
61            }
62            else
63            {
64               done = true;
65            }
66         }
67         return value;
68      }
69
```

Continued

Evaluator.java (cont.)

```
70      /**
71         Evaluates the next factor found in the expression.
72         @return the value of the factor
73      */
74      public int getFactorValue()
75      {
76         int value;
77         String next = tokenizer.peekToken();
78         if ("(".equals(next))
79         {
80            tokenizer.nextToken(); // Discard "("
81            value = getExpressionValue();
82            tokenizer.nextToken(); // Discard ")"
83         }
84         else
85         {
86            value = Integer.parseInt(tokenizer.nextToken());
87         }
88         return value;
89      }
90   }
```

ExpressionTokenizer.java

```
1  /**
2     This class breaks up a string describing an expression
3     into tokens: numbers, parentheses, and operators.
4  */
5  public class ExpressionTokenizer
6  {
7     private String input;
8     private int start; // The start of the current token
9     private int end; // The position after the end of the current token
10
11    /**
12       Constructs a tokenizer.
13       @param anInput the string to tokenize
14    */
15    public ExpressionTokenizer(String anInput)
16    {
17       input = anInput;
18       start = 0;
19       end = 0;
20       nextToken(); // Find the first token
21    }
22
```

Continued

ExpressionTokenizer.java (cont.)

```
23      /**
24         Peeks at the next token without consuming it.
25         @return the next token or null if there are no more tokens
26      */
27      public String peekToken()
28      {
29         if (start >= input.length()) { return null; }
30         else { return input.substring(start, end); }
31      }
32
```

Continued

ExpressionTokenizer.java (cont.)

```
33      /**
34         Gets the next token and moves the tokenizer to the following token.
35         @return the next token or null if there are no more tokens
36      */
37      public String nextToken()
38      {
39         String r = peekToken();
40         start = end;
41         if (start >= input.length()) { return r; }
42         if (Character.isDigit(input.charAt(start)))
43         {
44            end = start + 1;
45            while (end < input.length()
46                  && Character.isDigit(input.charAt(end)))
47            {
48               end++;
49            }
50         }
```

Continued

ExpressionTokenizer.java (cont.)

```
51          else
52          {
53             end = start + 1;
54          }
55          return r;
56       }
57    }
```

ExpressionCalculator.java

```
1  import java.util.Scanner;
2
3  /**
4     This program calculates the value of an expression
5     consisting of numbers, arithmetic operators, and parentheses.
6  */
7  public class ExpressionCalculator
8  {
9     public static void main(String[] args)
10    {
11       Scanner in = new Scanner(System.in);
12       System.out.print("Enter an expression: ");
13       String input = in.nextLine();
14       Evaluator e = new Evaluator(input);
15       int value = e.getExpressionValue();
16       System.out.println(input + "=" + value);
17    }
18 }
```

Program Run:

```
Enter an expression: 3+4*5
3+4*5=23
```

13.7 Backtracking

- Backtracking examines partial solutions, abandoning unsuitable ones and returning to consider other candidates.
- Can be used to
  - solve crossword puzzles.
  - escape from mazes.
  - find solutions to systems that are constrained by rules.

Backtracking Characteristic Properties

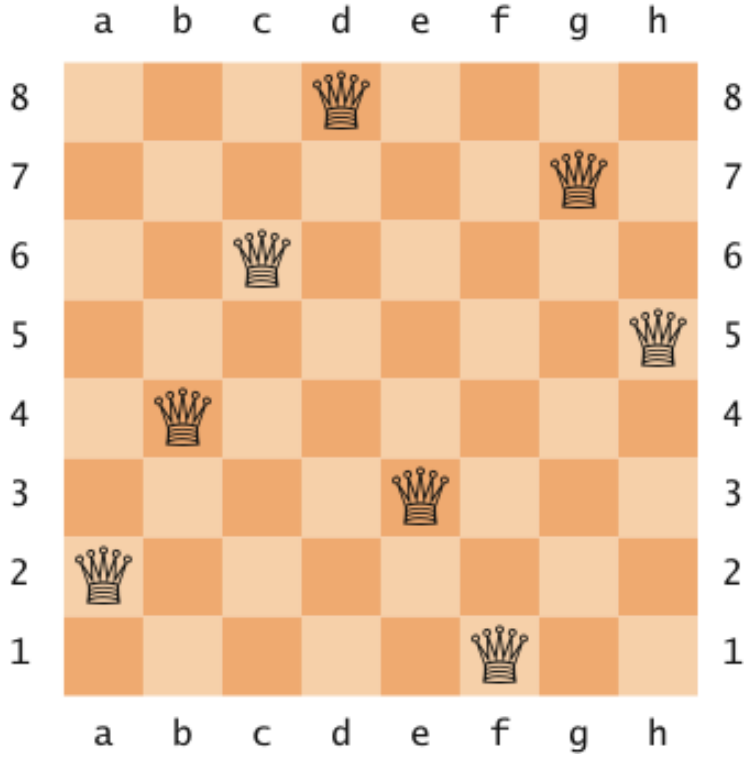
1. A procedure to examine a partial solution and determine whether to
   - Accept it as an actual solution or
   - Abandon it (because it either violates some rules or can never lead to a valid solution)
2. A procedure to extend a partial solution, generating one or more solutions that come closer to the goal

Recursive Backtracking Algorithm

```
Solve(partialSolution)
    Examine(partialSolution).
    If accepted
        Add partialSolution to the list of solutions.
    Else if not abandoned
        For each p in extend(partialSolution)
            Solve(p).
```

Eight Queens Problem (1)

- **Problem:** position eight queens on a chess board so that none of them attacks another according to the rules of chess
- A solution:



Eight Queens Problem (2)

- Easy to examine a partial solution:
  - If two queens attack one another, reject it
  - Otherwise, if it has eight queens, accept it
  - Otherwise, continue
- Easy to extend a partial solution:
  - Add another queen on an empty square
- Systematic extensions:
  - Place first queen on row 1
  - Place the next on row 2
  - Etc.

Class PartialSolution

```
public class PartialSolution
{
    private Queen[] queens;

    public int examine() { . . . }
    public PartialSolution[] extend() { . . . }
}
```

examine Method

```
public int examine()
{
   for (int i = 0; i < queens.length; i++)
   {
      for (int j = i + 1; j < queens.length; j++)
      {
         if (queens[i].attacks(queens[j])) { return ABANDON; }
      }
   }
   if (queens.length == NQUEENS) { return ACCEPT; }
   else { return CONTINUE; }
}
```

extend Method

```
public PartialSolution[] extend()
{
   // Generate a new solution for each column
   PartialSolution[] result = new PartialSolution[NQUEENS];
   for (int i = 0; i < result.length; i++)
   {
      int size = queens.length;
      // The new solution has one more row than this one
      result[i] = new PartialSolution(size + 1);
      // Copy this solution into the new one
      for (int j = 0; j < size; j++)
      {
         result[i].queens[j] = queens[j];
      }
      // Append the new queen into the ith column
      result[i].queens[size] = new Queen(size, i);
   }
   return result;
}
```

Diagonal Attack

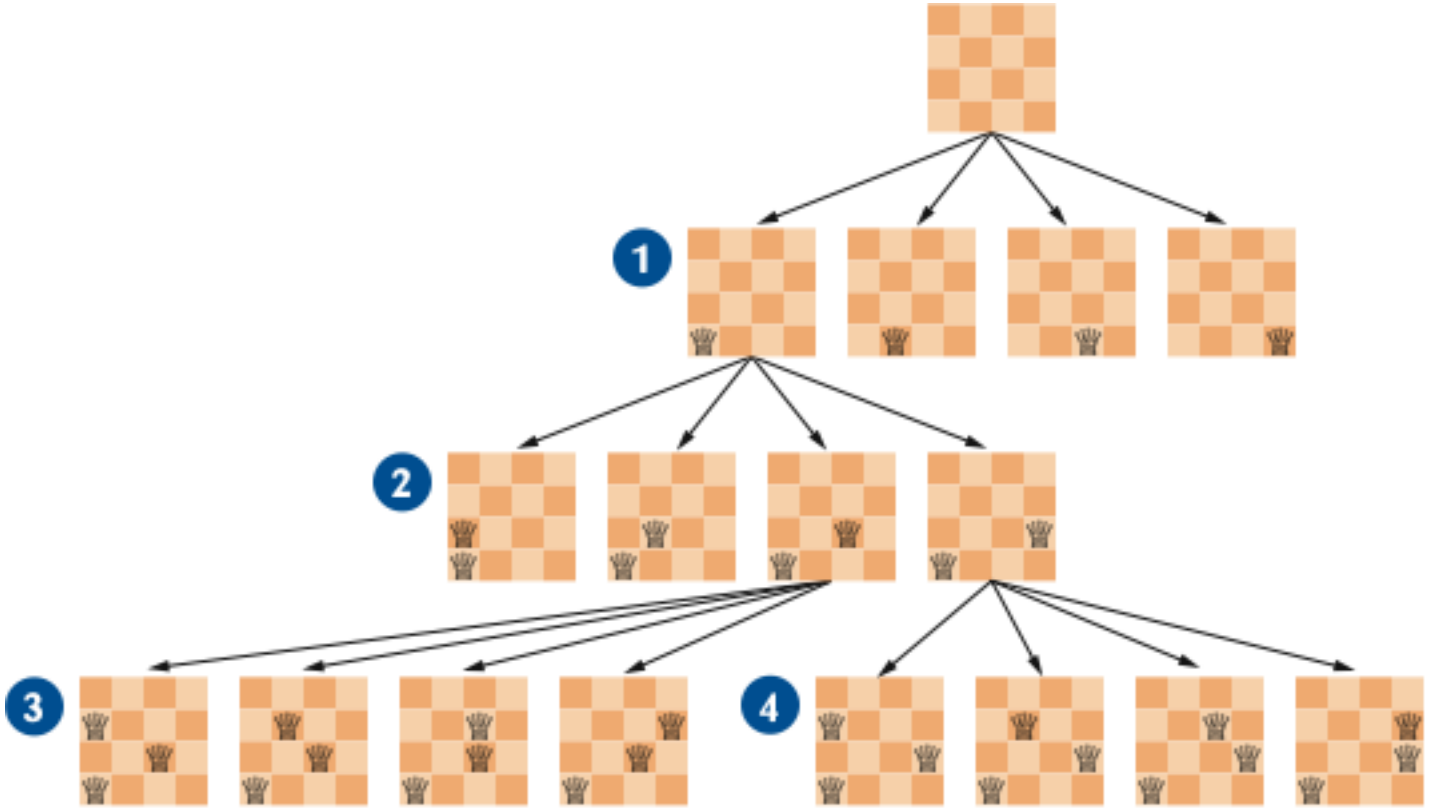
- To determine whether two queens attack each other diagonally:
  - Check whether slope is ±1

$$(row_2 - row_1)/(column_2 - column_1) = \pm 1$$

$$row_2 - row_1 = \pm(column_2 - column_1)$$

$$|row_2 - row_1| = |column_2 - column_1|$$

Backtracking in the Four Queens Problem (1)



Backtracking in the Four Queens Problem (2)

- Starting with a blank board, four partial solutions with a queen in row 1 (1)
- When the queen is in column 1, four partial solutions with a queen in row 2 (2)
  - Two are abandoned immediately
  - Other two lead to partial solutions with three queens (3) and (4), all but one of which are abandoned
- One partial solution is extended (5) to four queens, but all of those are abandoned as well

PartialSolution.java

```
1  /**
2     A partial solution to the eight queens puzzle.
3  */
4  public class PartialSolution
5  {
6     private Queen[] queens;
7     private static final int NQUEENS = 8;
8
9     public static final int ACCEPT = 1;
10    public static final int ABANDON = 2;
11    public static final int CONTINUE = 3;
12
13    /**
14       Constructs a partial solution of a given size.
15       @param size the size
16    */
17    public PartialSolution(int size)
18    {
19       queens = new Queen[size];
20    }
21
```

Continued

PartialSolution.java (cont.)

```
22    /**
23       Examines a partial solution.
24       @return one of ACCEPT, ABANDON, CONTINUE
25    */
26    public int examine()
27    {
28       for (int i = 0; i < queens.length; i++)
29       {
30          for (int j = i + 1; j < queens.length; j++)
31          {
32             if (queens[i].attacks(queens[j])) { return ABANDON; }
33          }
34       }
35       if (queens.length == NQUEENS) { return ACCEPT; }
36       else { return CONTINUE; }
37    }
38
```

Continued

PartialSolution.java (cont.)

```
39    /**
40       Yields all extensions of this partial solution.
41       @return an array of partial solutions that extend this solution.
42    */
43    public PartialSolution[] extend()
44    {
45       // Generate a new solution for each column
46       PartialSolution[] result = new PartialSolution[NQUEENS];
47       for (int i = 0; i < result.length; i++)
48       {
49          int size = queens.length;
50
51          // The new solution has one more row than this one
52          result[i] = new PartialSolution(size + 1);
53
54          // Copy this solution into the new one
55          for (int j = 0; j < size; j++)
56          {
57             result[i].queens[j] = queens[j];
58          }
59
```

Continued

PartialSolution.java (cont.)

```
60            // Append the new queen into the ith column
61            result[i].queens[size] = new Queen(size, i);
62         }
63         return result;
64      }
65
66      public String toString() { return Arrays.toString(queens); }
67   }
```

Queen.java

```
1  /**
2     A queen in the eight queens problem.
3  */
4  public class Queen
5  {
6     private int row;
7     private int column;
8
9     /**
10       Constructs a queen at a given position.
11       @param r the row
12       @param c the column
13    */
14    public Queen(int r, int c)
15    {
16       row = r;
17       column = c;
18    }
19
```

Continued

Queen.java (cont.)

```
20    /**
21       Checks whether this queen attacks another.
22       @param other the other queen
23       @return true if this and the other queen are in the same
24       row, column, or diagonal
25    */
26    public boolean attacks(Queen other)
27    {
28       return row == other.row
29          || column == other.column
30          || Math.abs(row - other.row) == Math.abs(column - other.column);
31    }
32
33    public String toString()
34    {
35       return "" + "abcdefgh".charAt(column) + (row + 1) ;
36    }
37 }
```

EightQueens.java

```
1  import java.util.Arrays;
2
3  /**
4     This class solves the eight queens problem using backtracking.
5  */
6  public class EightQueens
7  {
8     public static void main(String[] args)
9     {
10       solve(new PartialSolution(0));
11    }
12
```

Continued

EightQueens.java (cont.)

```
13    /**
14       Prints all solutions to the problem that can be extended from
15       a given partial solution.
16       @param sol the partial solution
17    */
18    public static void solve(PartialSolution sol)
19    {
20       int exam = sol.examine();
21       if (exam == PartialSolution.ACCEPT)
22       {
23          System.out.println(sol);
24       }
25       else if (exam != PartialSolution.ABANDON)
26       {
27          for (PartialSolution p : sol.extend())
28          {
29             solve(p);
30          }
31       }
32    }
33 }
```

Continued

EightQueens.java (cont.)

Program Run

```
[a1, e2, h3, f4, c5, g6, b7, d8]
[a1, f2, h3, c4, g5, d6, b7, e8]
[a1, g2, d3, f4, h5, b6, e7, c8]
 . . .
[f1, a2, e3, b4, h5, c6, g7, d8]
 . . .
[h1, c2, a3, f4, b5, e6, g7, d8]
[h1, d2, a3, c4, f5, b6, g7, e8]
```

(92 solutions)

Summary

Control Flow in a Recursive Computation

- A recursive computation solves a problem by using the solution to the same problem with simpler inputs.
- For a recursion to terminate, there must be special cases for the simplest values.

Design a Recursive Solution to a Problem

Summary

Identify Recursive Helper Methods for Solving a Problem

- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

Contrast the Efficiency of Recursive and Non-Recursive Algorithms

- Occasionally, a recursive solution runs much slower than its iterative counterpart. However, in most cases, the recursive solution is only slightly slower.
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

Summary

Review a Complex Recursion Example That Cannot Be Solved with a Simple Loop

- The permutations of a string can be obtained more naturally through recursion than with a loop.

Recognize the Phenomenon of Mutual Recursion in an Expression Evaluator

- In a mutual recursion, a set of cooperating methods calls each other repeatedly.

Summary

Use Backtracking to Solve Problems That Require Trying Out Multiple Paths

- Backtracking examines partial solutions, abandoning unsuitable ones and returning to consider other candidates.