CHAPTER 13

RECURSION

Slides by Rick Giles

Chapter Goals

- To learn to "think recursively"
- To be able to use recursive helper methods
- To understand the relationship between recursion and iteration
- To understand when the use of recursion affects the efficiency of an algorithm
- To analyze problems that are much easier to solve by recursion than by iteration
- To process data with recursive structures using mutual recursion

Contents

13.1 Triangle Numbers Revisited

- Triangle shape of side length 4:

```
[]
[][]
[][][]
[][][][]
```

- Will use recursion to compute the area of a triangle of width *n* , assuming each [] square has an area of 1
- Also called the n^{th} *triangle number*
- The third triangle number is 6, the fourth is 10

Outline of `Triangle` Class

```
public class Triangle
{
   private int width;
   public Triangle(int aWidth)
   {
      width = aWidth;
   }
   public int getArea()
   {
      ...
   }
}
```

Handling Triangle of Width 1

- The triangle consists of a single square
- Its area is 1
- Take care of this case first:

```
public int getArea()
{
   if (width == 1) { return 1; }
   . . .
}
```

Handling The General Case

- Assume we know the area of the smaller, colored triangle:

```
[]
[][]
[][][]
[][][][]
```

- Area of larger triangle can be calculated as

```
smallerArea + width
```

- To get the area of the smaller triangle
  - *Make a smaller triangle and ask it for its area:*

```
Triangle smallerTriangle = new Triangle(width - 1);
int smallerArea = smallerTriangle.getArea();
```

Completed `getArea` Method

```
public int getArea()
{
   if (width == 1) { return 1; }
   Triangle smallerTriangle = new Triangle(width - 1);
   int smallerArea = smallerTriangle.getArea();
   return smallerArea + width;
}
```

Computing the Area of a Triangle With Width 4

- `getArea` method makes a smaller triangle of width 3
- It calls `getArea` on that triangle
  - That method makes a smaller triangle of width 2
  - It calls `getArea` on that triangle
    - That method makes a smaller triangle of width 1
    - It calls `getArea` on that triangle
      - That method returns 1
    - The method returns `smallerArea + width` = 1 + 2 = 3
  - The method returns `smallerArea + width` = 3 + 3 = 6
- The method returns `smallerArea + width` = 6 + 4 = 10

Recursive Computation

- A **recursive computation** solves a problem by using the solution to the same problem with simpler inputs
- Call pattern of a **recursive method** is complicated
  - Key: *Don’t think about it*

Successful Recursion

- Every recursive call must simplify the computation in some way
- There must be special cases to handle the simplest computations directly

Other Ways to Compute Triangle Numbers

- The area of a triangle equals the sum:

```
1 + 2 + 3 + ... + width
```

- Using a simple loop:

```
double area = 0;
for (int i = 1; i <= width; i++)
   area = area + i;
```

- Using math:

$$1 + 2 + \ldots + n = n \times (n + 1)/2$$

=> `width * (width + 1) / 2`

Triangle.java

```
/**
   A triangular shape composed of stacked unit squares like this:
   []
   [][]
   [][][]
   ...
*/
public class Triangle
{
   private int width;

   /**
      Constructs a triangular shape.
      @param aWidth the width (and height) of the triangle
   */
   public Triangle(int aWidth)
   {
      width = aWidth;
   }

```

Continued

Triangle.java (cont.)

```java
    /**
       Computes the area of the triangle.
       @return the area
    */
    public int getArea()
    {
       if (width <= 0) { return 0; }
       if (width == 1) { return 1; }
       else
       {
          Triangle smallerTriangle = new Triangle(width - 1);
          int smallerArea = smallerTriangle.getArea();
          return smallerArea + width;
       }
    }
}
```

TriangleTester.java

```
public class TriangleTester
{
   public static void main(String[] args)
   {
      Triangle t = new Triangle(10);
      int area = t.getArea();
      System.out.println("Area: " + area);
      System.out.println("Expected: 55");
   }
}
```

Program Run:

```
Area: 55
Expected: 55
```

13.2 Problem Solving: Thinking Recursively

- 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

```
/**
    Tests whether a text is a palindrome.
    @param text a string that is being checked
    @return true if text is a palindrome, false otherwise
*/
public static boolean isPalindrome(String Text)
{
    . . .
}
```

Thinking Recursively: Step 1

- Consider various ways to simplify inputs.
- 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: Step 2 (1)

- Combine solutions with simpler inputs into a solution of the original problem.
- Most promising simplification: *Remove both 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*

Thinking Recursively: Step 2 (2)

- 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: Step 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: Step 4 (1)

- Implement the solution by combining the simple cases and the reduction step.

```
public static boolean isPalindrome(String text)
{
   int length = text.length();
   // Separate case for shortest strings.
   if (length <= 1) { return true; }
   else
   {
      // 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: Step 4 (2)

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

Continued

Thinking Recursively: Step 4 (3)

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

13.3 Recursive Helper Methods

- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.
- Consider the palindrome test of previous section.
- It is a bit inefficient to construct new string objects in every step.

Substring Palindromes (1)

- Rather than testing whether the sentence is a palindrome, check whether a substring is a palindrome:

```
/**
   Tests whether a substring is a palindrome.
   @param text a string that is being checked
   @param start the index of the first character of the substring
   @param end the index of the last character of the substring
   @return true if the substring is a palindrome
*/
public static boolean isPalindrome(String text, int start, int end)
```

Substring Palindromes (2)

- Then, simply call the helper method with positions that test the entire string:

```
public static boolean isPalindrome(String text)
{
   return isPalindrome(text, 0, text.length() - 1);
}
```

Recursive Helper Method `isPalindrome` (1)

```
public static boolean isPalindrome(String text, int start, int end)
{
   // Separate case for substrings of length 0 and 1.
   if (start >= end) { return true; }
   else
   {
      // Get first and last characters, converted to lowercase.
      char first = Character.toLowerCase(text.charAt(start));
      char last = Character.toLowerCase(text.charAt(end));
      if (Character.isLetter(first) && Character.isLetter(last))
      {
         if (first == last)
         {
            // Test substring that doesn't contain the matching letters.
            return isPalindrome(text, start + 1, end - 1);
         }
         else
         {
            return false;
         }
```

Continued

Recursive Helper Method `isPalindrome` (2)

```
      }
      else if (!Character.isLetter(last))
      {
         // Test substring that doesn't contain the last character.
         return isPalindrome(text, start, end - 1);
      }
      else
      {
         // Test substring that doesn't contain the first character.
         return isPalindrome(text, start + 1, end);
      }
   }
}
```

13.4 The Efficiency of Recursion

- Fibonacci sequence:
  Sequence of numbers defined by

  $f_1 = 1$
  $f_2 = 1$
  $f_n = f_{n-1} + f_{n-2}$

- First ten terms:

  1, 1, 2, 3, 5, 8, 13, 21, 34, 55

RecursiveFib.java

```
import java.util.Scanner;

/**
   This program computes Fibonacci numbers using a recursive method.
*/
public class RecursiveFib
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter n: ");
      int n = in.nextInt();

      for (int i = 1; i <= n; i++)
      {
         long f = fib(i);
         System.out.println("fib(" + i + ") = " + f);
      }
   }

```

Continued

RecursiveFib.java (cont.)

```java
   /**
      Computes a Fibonacci number.
      @param n an integer
      @return the nth Fibonacci number
   */
   public static long fib(int n)
   {
      if (n <= 2) { return 1; }
      else return fib(n - 1) + fib(n - 2);
   }
}
```

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
```

Efficiency of Recursion

- Recursive implementation of `fib` is straightforward.
- Watch the output closely as you run the test program.
- First few calls to `fib` are quite fast.
- For larger values, the program pauses an amazingly long time between outputs.
- To find out the problem, let's insert **trace messages**.

RecursiveFibTracer.java

```java
import java.util.Scanner;

/**
    This program prints trace messages that show how often the
    recursive method for computing Fibonacci numbers calls itself.
*/
public class RecursiveFibTracer
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter n: ");
      int n = in.nextInt();

      long f = fib(n);

      System.out.println("fib(" + n + ") = " + f);
   }

```

Continued

RecursiveFibTracer.java (cont.)

```
    /**
       Computes a Fibonacci number.
       @param n an integer
       @return the nth Fibonacci number
    */
    public static long fib(int n)
    {
       System.out.println("Entering fib: n = " + n);
       long f;
       if (n <= 2) { f = 1; }
       else { f = fib(n - 1) + fib(n - 2); }
       System.out.println("Exiting fib: n = " + n
             + " return value = " + f);
       return f;
    }
 }
```

Continued

RecursiveFibTracer.java (cont.)

Program Run:

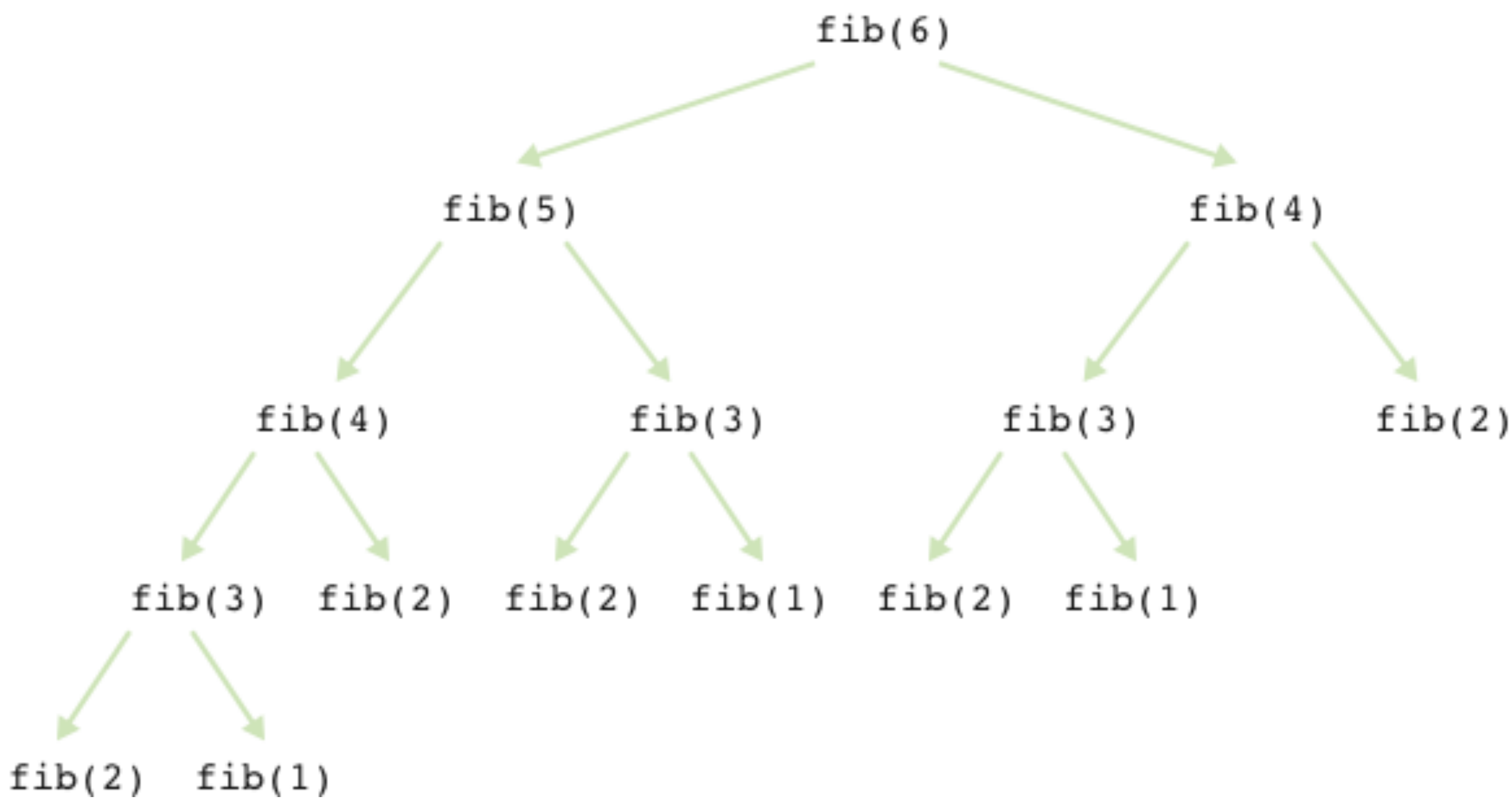
```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
```

Continued

RecursiveFibTracer.java (cont.)

```
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Exiting fib: n = 5 return value = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Exiting fib: n = 6 return value = 8
fib(6) = 8
```

Call Pattern of Recusive `fib` Method



Efficiency of Recursion

- Method takes so long because it computes the same values over and over.
- Computation of `fib(6)` calls `fib(3)` three times.
- Imitate the pencil-and-paper process to avoid computing the values more than once.

LoopFib.java

```java
import java.util.Scanner;

/**
   This program computes Fibonacci numbers using an iterative method.
*/
public class LoopFib
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter n: ");
      int n = in.nextInt();

      for (int i = 1; i <= n; i++)
      {
         long f = fib(i);
         System.out.println("fib(" + i + ") = " + f);
      }
   }

```

Continued

LoopFib.java (cont.)

```java
   /**
      Computes a Fibonacci number.
      @param n an integer
      @return the nth Fibonacci number
   */
   public static long fib(int n)
   {
      if (n <= 2) { return 1; }
      else
      {
         long olderValue = 1;
         long oldValue = 1;
         long newValue = 1;
         for (int i = 3; i <= n; i++)
         {
            newValue = oldValue + olderValue;
            olderValue = oldValue;
            oldValue = newValue;
         }
         return newValue;
      }
   }
}
```

Continued

LoopFib.java (cont.)

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
```

Efficiency of Recursion

- Occasionally, a recursive solution runs much slower than its iterative counterpart.
- In most cases, the recursive solution is only slightly slower.
- The iterative `isPalindrome` performs only slightly better than recursive solution.
  - *Each recursive method call takes a certain amount of processor time*

Efficiency of Recursion

- Smart compilers can avoid recursive method calls if they follow simple patterns.
- Most compilers don’t do that
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution .
- “To iterate is human, to recurse divine.”
  - L. Peter Deutsch

Iterative `isPalindrome` Method

```
public static boolean isPalindrome(String text)
{
   int start = 0;
   int end = text.length() - 1;
   while (start < end)
   {
      char first = Character.toLowerCase(text.charAt(start));
      char last = Character.toLowerCase(text.charAt(end));
      if (Character.isLetter(first) && Character.isLetter(last))
      {
         // Both are letters.
         if (first == last)
         {
            start++;
            end--;
         }
         else { return false; }
      }
      if (!Character.isLetter(last)) { end--; }
      if (!Character.isLetter(first)) { start++; }
   }
   return true;
}
```

13.5 Permutations

- Design a class that will list all permutations of a string, where a **permutation** is a rearrangement of the letters
- The string `"eat"` has six permutations:

```
"eat"
"eta"
"aet"
"ate"
"tea"
"tae"
```

Generate All Permutations (1)

- Generate all permutations that start with `'e'`, then `'a'`, then `'t'`
- The string `"eat"` has six permutations:

```
"eat"
"eta"
"aet"
"ate"
"tea"
"tae"
```

Generate All Permutations (2)

- Generate all permutations that start with `'e'`, then `'a'`, then `'t'`
- To generate permutations starting with `'e'`, we need to find all permutations of `"at"`
- This is the same problem with simpler inputs
- Use recursion

Implementing `permutations` Method

- Loop through all positions in the word to be permuted
- For each of them, compute the shorter word obtained by removing the `i`th letter:

```
String shorter = word.substring(0, i) + word.substring(i + 1);
```

- Compute the permutations of the shorter word:

```
ArrayList<String> shorterPermutations = permutations(shorter);
```

Implementing `permutations` Method

- Add the removed letter from to the front of all permutations of the shorter word:

```
for (String s : shorterPermutations)
{
   result.add(word.charAt(i) + s);
}
```

- Special case for the simplest string, the empty string, which has a single permutation - itself

Permutations.java

```
import java.util.ArrayList;

/**
    This class computes permutations of a string.
*/
public class Permutations
{
   public static void main(String[] args)
   {
      for (String s : permutations(“eat”))
      {
         System.out.println(s);
      }
   }

```

Continued

Permutations.java (cont.)

```
/**
   Gets all permutations of a given word.
   @param  word the string to permute
   @return  a list of all permutations
*/
public static ArrayList<String> permutations(String word)
{
    ArrayList<String> result = new ArrayList<String>();

    // The empty string has a single permutation: itself
    if (word.length() == 0)
    {
        result.add(word);
        return result;
    }
```

Continued

Permutations.java (cont.)

```
    else
    {
        // Loop through all character positions
        for (int i = 0; i < word.length(); i++)
        {
            // Form a shorter word by removing the ith character
            String shorter = word.substring(0, i) + word.substring(i + 1);

            // Generate all permutations of the simpler word
            ArrayList<String> shorterPermutations = permutations(shorter);

            // Add the removed character to the front of
            // each permutation of the simpler word
            for (String s : shorterPermutations)
            {
                result.add(word.charAt(i) + s);
            }
        }
        // Return all permutations
        return result;
    }
}
```

Continued

Permutations.java (cont.)

Program Run:

```
eat
eta
aet
ate
tea
tae
```

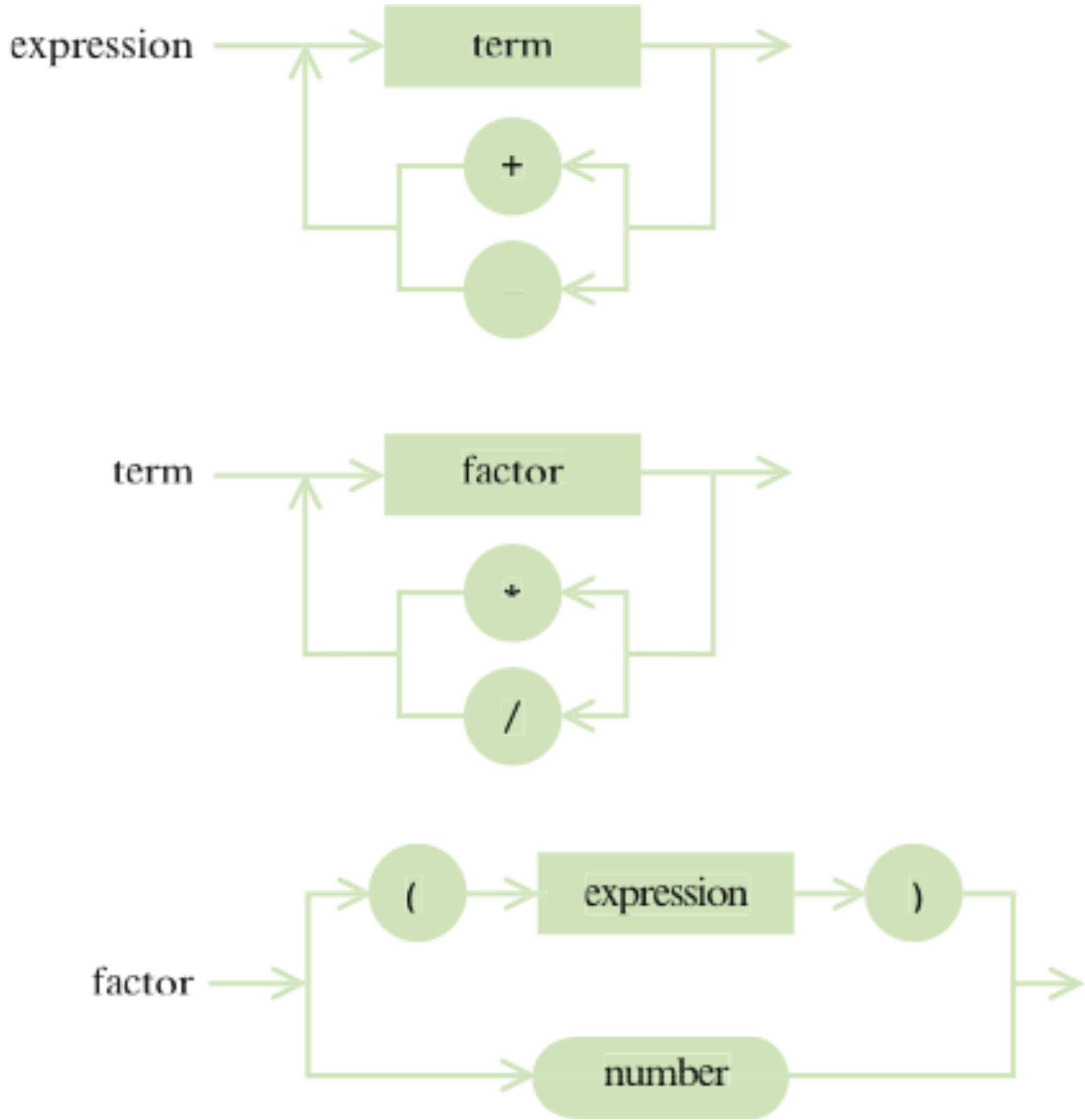
13.6 Mutual Recursion

- **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

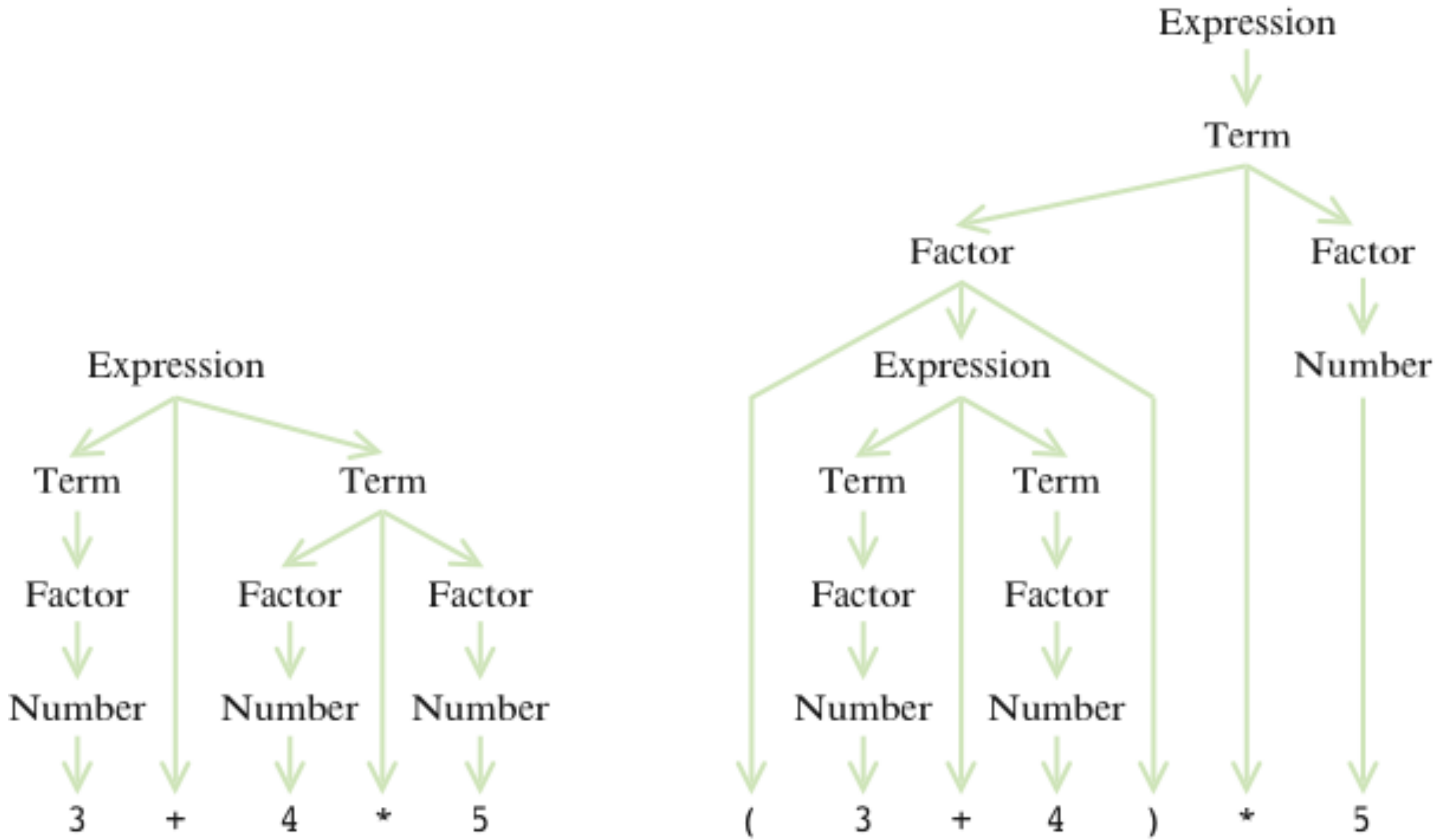
Syntax Diagrams for Evaluating an Expression



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

```
/**
    A class that can compute the value of an arithmetic expression.
*/
public class Evaluator
{
   private ExpressionTokenizer tokenizer;

   /**
      Constructs an evaluator.
      @param anExpression a string containing the expression
      to be evaluated
   */
   public Evaluator(String anExpression)
   {
      tokenizer = new ExpressionTokenizer(anExpression);
   }

```

Continued

Evaluator.java (cont.)

```
/**
   Evaluates the expression.
   @return the value of the expression.
*/
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;
}

```

Continued

Evaluator.java (cont.)

```
/**
    Evaluates the next term found in the expression.
    @return the value of the term
*/
public int getTermValue()
{
    int value = getFactorValue();
    boolean done = false;
    while (!done)
    {
        String next = tokenizer.peekToken();
        if ("*".equals(next) || "/".equals(next))
        {
            tokenizer.nextToken();
            int value2 = getFactorValue();
            if ("*".equals(next)) { value = value * value2; }
            else { value = value / value2; }
        }
        else
        {
            done = true;
        }
    }
    return value;
}
```

Continued

Evaluator.java (cont.)

```
    /**
       Evaluates the next factor found in the expression.
       @return the value of the factor
    */
    public 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;
    }
 }
```

ExpressionTokenizer.java

```
/**
    This class breaks up a string describing an expression
    into tokens: numbers, parentheses, and operators.
*/
public class ExpressionTokenizer
{
   private String input;
   private int start; // The start of the current token
   private int end; // The position after the end of the current token

   /**
      Constructs a tokenizer.
      @param anInput the string to tokenize
   */
   public ExpressionTokenizer(String anInput)
   {
      input = anInput;
      start = 0;
      end = 0;
      nextToken(); // Find the first token
   }

```

Continued

ExpressionTokenizer.java (cont.)

```
   /**
      Peeks at the next token without consuming it.
      @return the next token or null if there are no more tokens
   */
   public String peekToken()
   {
      if (start >= input.length()) { return null; }
      else { return input.substring(start, end); }
   }

```

Continued

ExpressionTokenizer.java (cont.)

```
   /**
      Gets the next token and moves the tokenizer to the following token.
      @return the next token or null if there are no more tokens
   */
   public String nextToken()
   {
      String r = peekToken();
      start = end;
      if (start >= input.length()) { return r; }
      if (Character.isDigit(input.charAt(start)))
      {
         end = start + 1;
         while (end < input.length()
               && Character.isDigit(input.charAt(end)))
         {
            end++;
         }
      }
```

Continued

ExpressionTokenizer.java (cont.)

```java
      else
      {
         end = start + 1;
      }
      return r;
   }
}
```

ExpressionCalculator.java

```java
import java.util.Scanner;

/**
    This program calculates the value of an expression
    consisting of numbers, arithmetic operators, and parentheses.
*/
public class ExpressionCalculator
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter an expression: ");
      String input = in.nextLine();
      Evaluator e = new Evaluator(input);
      int value = e.getExpressionValue();
      System.out.println(input + "=" + value);
   }
}
```

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

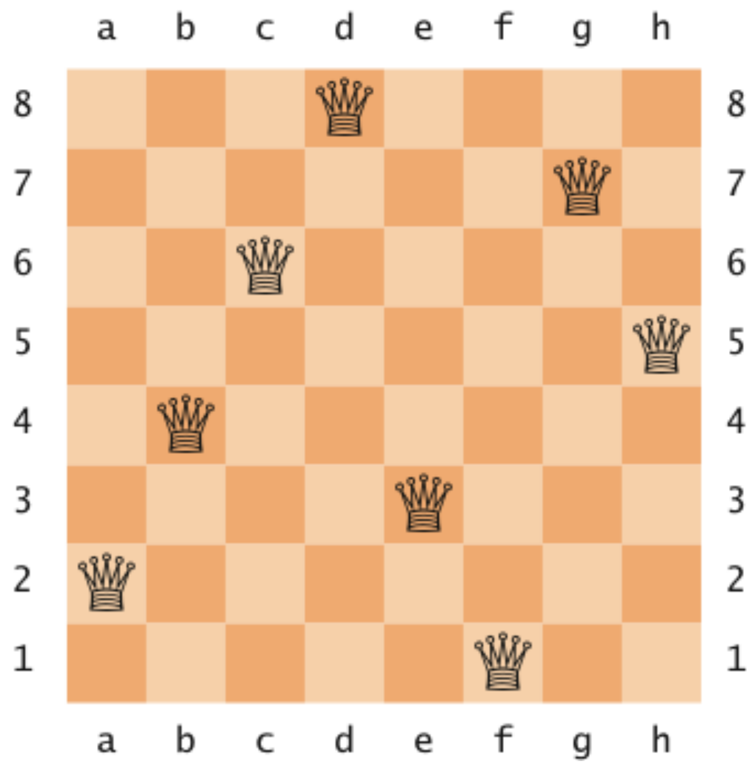
1. A procedure to examine a partial solution and determine whether to
   - Accept it as an actual solution or
   - Abandon it (because it either violates some rules or can never lead to a valid solution)
2. A procedure to extend a partial solution, generating one or more solutions that come closer to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

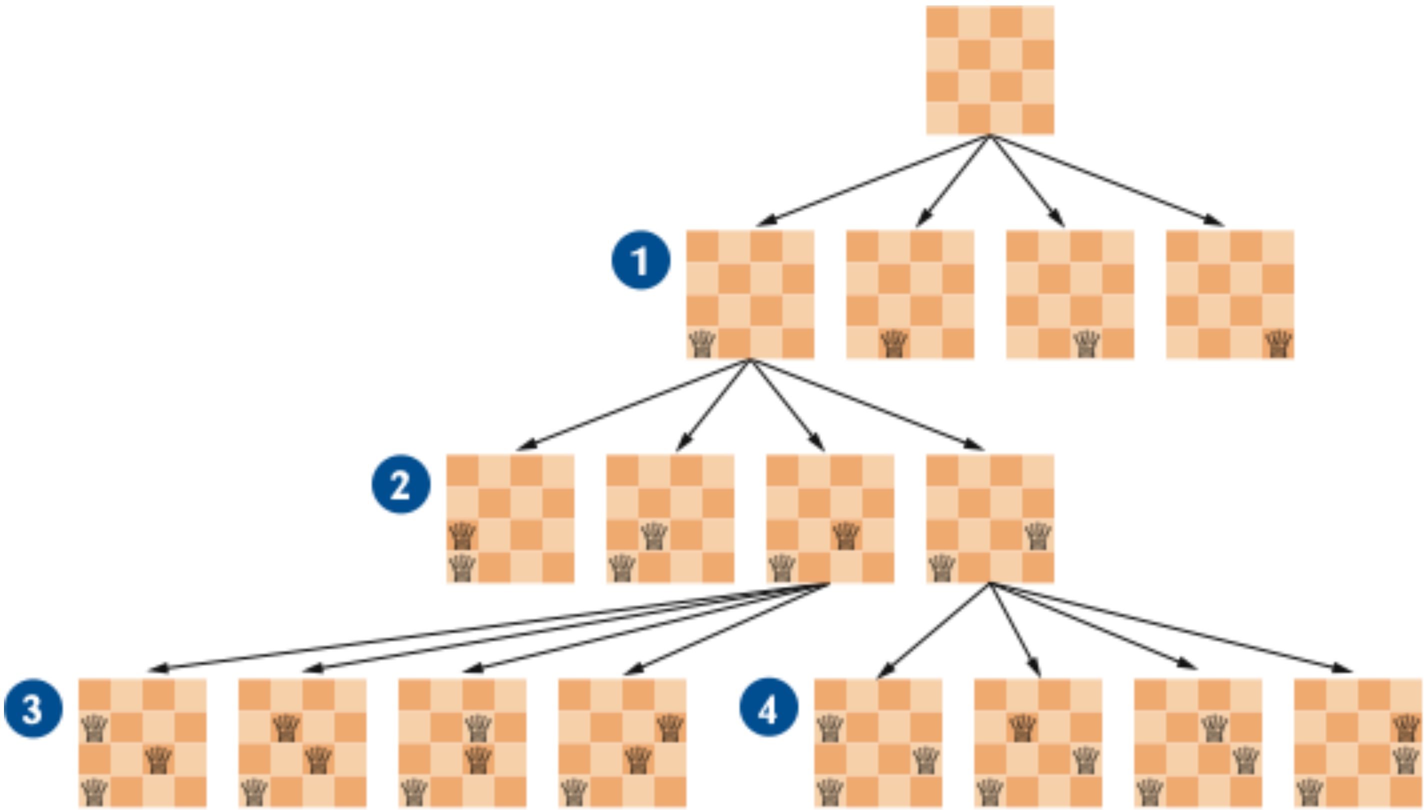
- To determine whether two queens attack each other diagonally:
  - Check whether slope is ± 1

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

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

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

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

PartialSolution.java

```java
/**
   A partial solution to the eight queens puzzle.
*/
public class PartialSolution
{
   private Queen[] queens;
   private static final int NQUEENS = 8;

   public static final int ACCEPT = 1;
   public static final int ABANDON = 2;
   public static final int CONTINUE = 3;

   /**
      Constructs a partial solution of a given size.
      @param size the size
   */
   public PartialSolution(int size)
   {
      queens = new Queen[size];
   }

```

Continued

PartialSolution.java (cont.)

```
   /**
      Examines a partial solution.
      @return one of ACCEPT, ABANDON, CONTINUE
   */
   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; }
   }
```

Continued

PartialSolution.java (cont.)

```java
/**
   Yields all extensions of this partial solution.
   @return an array of partial solutions that extend this solution.
*/
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];
      }

```

Continued

PartialSolution.java (cont.)

```java
            // Append the new queen into the ith column
            result[i].queens[size] = new Queen(size, i);
        }
        return result;
    }

    public String toString() { return Arrays.toString(queens); }
}
```

Queen.java

```java
/**
   A queen in the eight queens problem.
*/
public class Queen
{
   private int row;
   private int column;

   /**
      Constructs a queen at a given position.
      @param r the row
      @param c the column
   */
   public Queen(int r, int c)
   {
      row = r;
      column = c;
   }

```

Continued

Queen.java (cont.)

```java
    /**
        Checks whether this queen attacks another.
        @param other the other queen
        @return true if this and the other queen are in the same
        row, column, or diagonal
    */
    public boolean attacks(Queen other)
    {
        return row == other.row
            || column == other.column
            || Math.abs(row - other.row) == Math.abs(column - other.column);
    }

    public String toString()
    {
        return "" + "abcdefgh".charAt(column) + (row + 1) ;
    }
}
```

EightQueens.java

```java
import java.util.Arrays;

/**
    This class solves the eight queens problem using backtracking.
*/
public class EightQueens
{
   public static void main(String[] args)
   {
      solve(new PartialSolution(0));
   }

```

Continued

EightQueens.java (cont.)

```
    /**
       Prints all solutions to the problem that can be extended from
       a given partial solution.
       @param sol the partial solution
    */
    public static void solve(PartialSolution sol)
    {
       int exam = sol.examine();
       if (exam == PartialSolution.ACCEPT)
       {
          System.out.println(sol);
       }
       else if (exam != PartialSolution.ABANDON)
       {
          for (PartialSolution p : sol.extend())
          {
             solve(p);
          }
       }
    }
}
```

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.