RECURSION

Chapter 5

Chapter Objectives

- To understand how to think recursively
- To learn how to trace a recursive method
- To learn how to write recursive algorithms and methods for searching arrays
- To learn about recursive data structures and recursive methods for a `LinkedList` class
- To understand how to use recursion to solve the Towers of Hanoi problem
- To understand how to use recursion to process two-dimensional images
- To learn how to apply backtracking to solve search problems such as finding a path through a maze

Recursion

- Recursion can solve many programming problems that are difficult to conceptualize and solve linearly
- In the field of artificial intelligence, recursion often is used to write programs that exhibit intelligent behavior:
  - playing games of chess
  - proving mathematical theorems
  - recognizing patterns, and so on
- Recursive algorithms can
  - compute factorials
  - compute a greatest common divisor
  - process data structures (strings, arrays, linked lists, etc.)
  - search efficiently using a binary search
  - find a path through a maze, and more

Recursive Thinking

Section 5.1

Recursive Thinking

- Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that are difficult to solve by other means
- Recursion reduces a problem into one or more simpler versions of itself

Recursive Thinking (cont.)

Recursive Algorithm to Process Nested Figures

```
if there is one figure
    do whatever is required to the figure
else
    do whatever is required to the outer figure
    process the figures nested inside the outer figure
in the same way
```

Recursive Thinking (cont.)

- Consider searching for a target value in an array
  - Assume the array elements are sorted in increasing order
  - We compare the target to the middle element and, if the middle element does not match the target, search either the elements before the middle element or the elements after the middle element
  - Instead of searching n elements, we search $n/2$ elements

Recursive Thinking (cont.)

Recursive Algorithm to Search an Array

```
if the array is empty
    return -1 as the search result
else if the middle element matches the target
    return the subscript of the middle element as the result
else if the target is less than the middle element
    recursively search the array elements preceding the middle element and return the result
else
    recursively search the array elements following the middle element and return the result
```

Steps to Design a Recursive Algorithm

- There must be at least one case (the base case), for a small value of *n*, that can be solved directly
- A problem of a given size *n* can be reduced to one or more smaller versions of the same problem (recursive case(s))
- Identify the base case(s) and solve it/them directly
- Devise a strategy to reduce the problem to smaller versions of itself while making progress toward the base case
- Combine the solutions to the smaller problems to solve the larger problem

Recursive Algorithm for Finding the Length of a String

```
if the string is empty (has no characters)
    the length is 0
else
    the length is 1 plus the length of the string that
    excludes the first character
```

Recursive Algorithm for Finding the Length of a String (cont.)

```
/** Recursive method length
    @param str The string
    @return The length of the string
*/
public static int length(String str) {
  if (str == null || str.equals(""))
  return 0;
else
  return 1 + length(str.substring(1));
}
```

Recursive Algorithm for Printing String Characters

```
/** Recursive method printChars
post: The argument string is displayed, one character
per line
@param str The string
*/
public static void printChars(String str) {
  if (str == null || str.equals(""))
    return;
  else {
    System.out.println(str.charAt(0));
    printChars(str.substring(1));
  }
}
```

Recursive Algorithm for Printing String Characters in Reverse

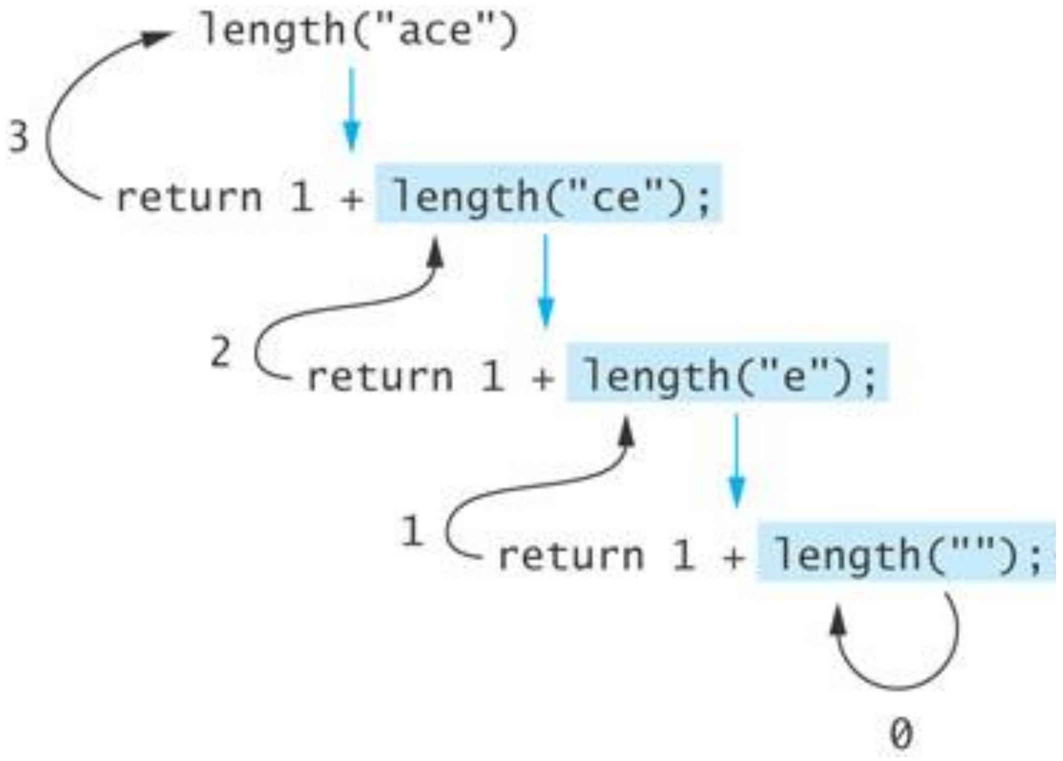
```
/** Recursive method printCharsReverse
post: The argument string is displayed in reverse, one
character per line
@param str The string
*/
public static void printCharsReverse(String str) {
  if (str == null || str.equals(""))
    return;
  else {
    printCharsReverse(str.substring(1));
    System.out.println(str.charAt(0));
  }
}
```

Proving that a Recursive Method is Correct

- Proof by induction
  - Prove the theorem is true for the base case
  - Show that if the theorem is assumed true for n, then it must be true for n+1
- Recursive proof is similar to induction
  - Verify the base case is recognized and solved correctly
  - Verify that each recursive case makes progress towards the base case
  - Verify that if all smaller problems are solved correctly, then the original problem also is solved correctly

Tracing a Recursive Method

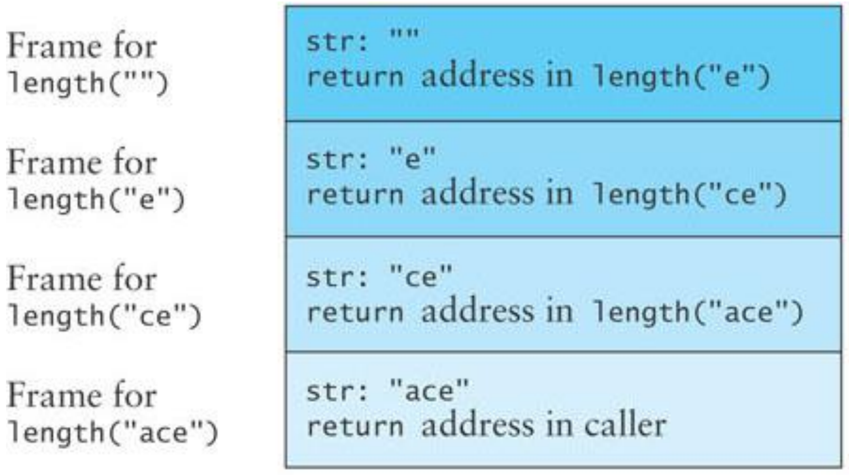
- The process of returning from recursive calls and computing the partial results is called *unwinding the recursion*



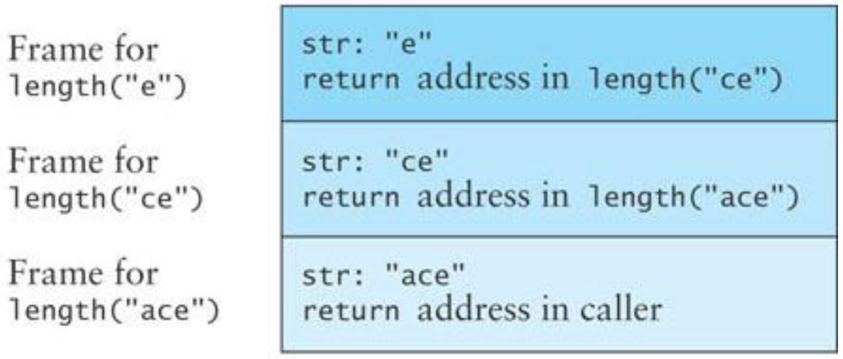
Run-Time Stack and Activation Frames

- Java maintains a run-time stack on which it saves new information in the form of an *activation frame*
- The activation frame contains storage for
  - method arguments
  - local variables (if any)
  - the return address of the instruction that called the method
- Whenever a new method is called (recursive or not), Java pushes a new activation frame onto the run-time stack

Run-Time Stack and Activation Frames (cont.)



Run-time stack after all calls



Run-time stack after return from last call

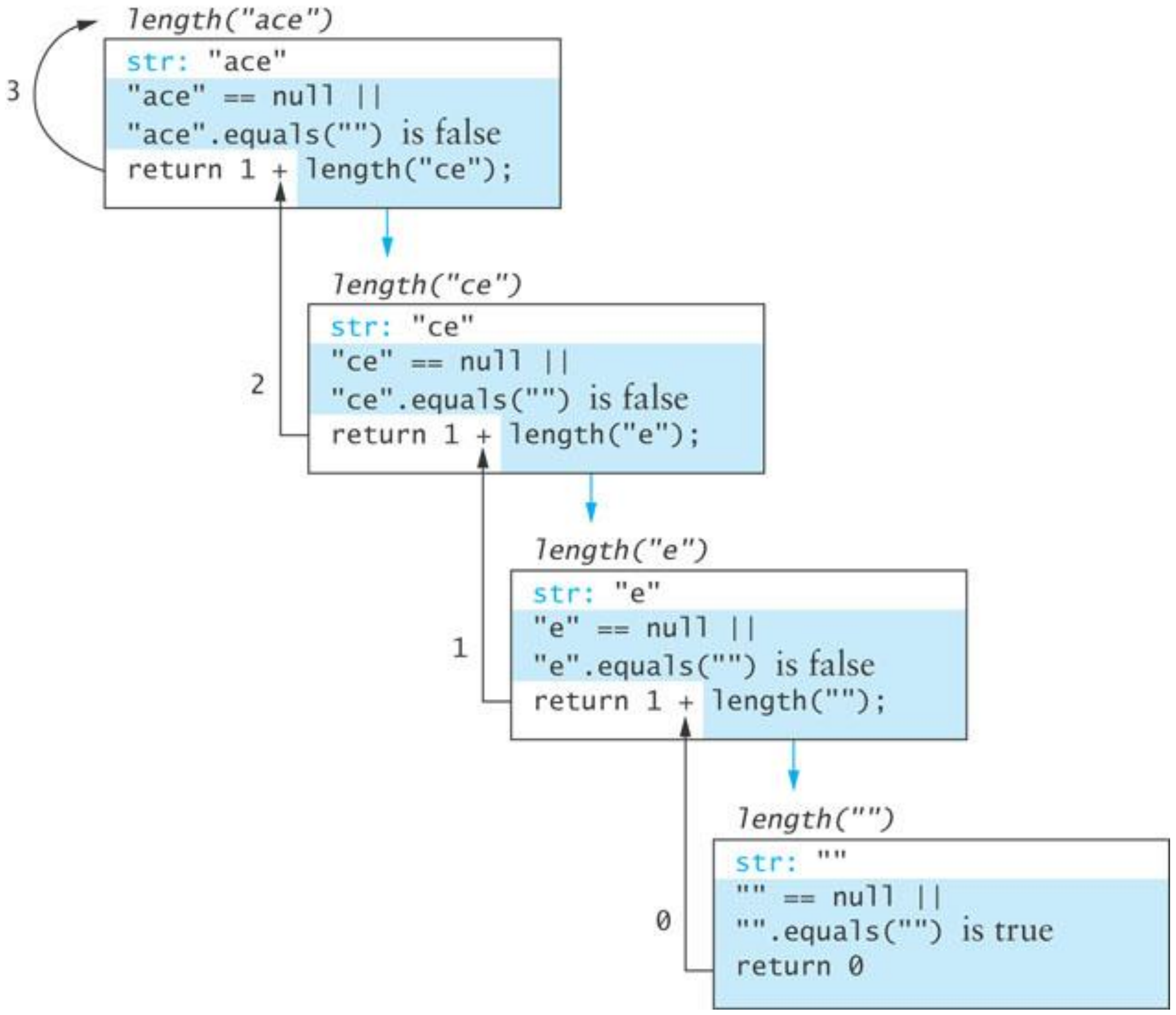
Analogy for the Run-Time Stack for Recursive Calls

- An office tower has an employee on each level each with the same list of instructions
  - The employee on the bottom level carries out part of the instructions, calls the employee on the next level up and is put on hold
    - The employee on the next level completes part of the instructions and calls the employee on the next level up and is put on hold
      - The employee on the next level completes part of the instructions and calls the employee on the next level up and is put on hold
        - The employee on the next level completes part of the instructions and calls the employee on the next level up and is put on hold, an so on until the top level is reached

Analogy for the Run-Time Stack for Recursive Calls (cont.)

- When the employee on the top level finishes the instructions, that employee returns an answer to the employee below
  - The employee below resumes, and when finished, returns an answer to the employee below
    - The employee below resumes, and when finished, returns an answer to the employee below
      - The employee below resumes, and when finished, returns an answer to the employee below, and so on
- Eventually the bottom is reached, and all instructions are executed

Run-Time Stack and Activation Frames



Recursive Definitions of Mathematical Formulas

Section 5.2

Recursive Definitions of Mathematical Formulas

- Mathematicians often use recursive definitions of formulas that lead naturally to recursive algorithms
- Examples include:
  - factorials
  - powers
  - greatest common divisors (gcd)

Factorial of *n*: *n*!

- The factorial of *n*, or *n*! is defined as follows:

  $0! = 1$

  $n! = n \times (n - 1)!\ (n > 0)$

- The base case: *n* is equal to 0
- The second formula is a recursive definition

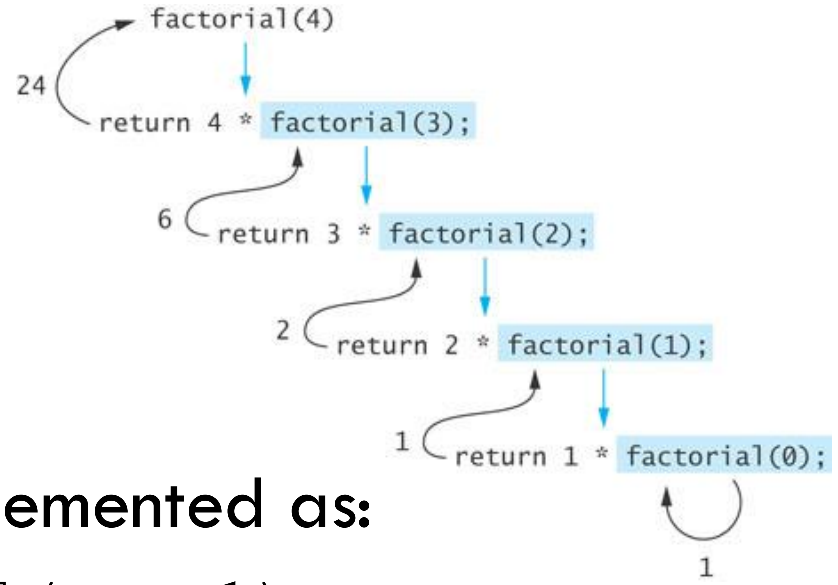
Factorial of *n*: *n*! (cont.)

- The recursive definition can be expressed by the following algorithm:

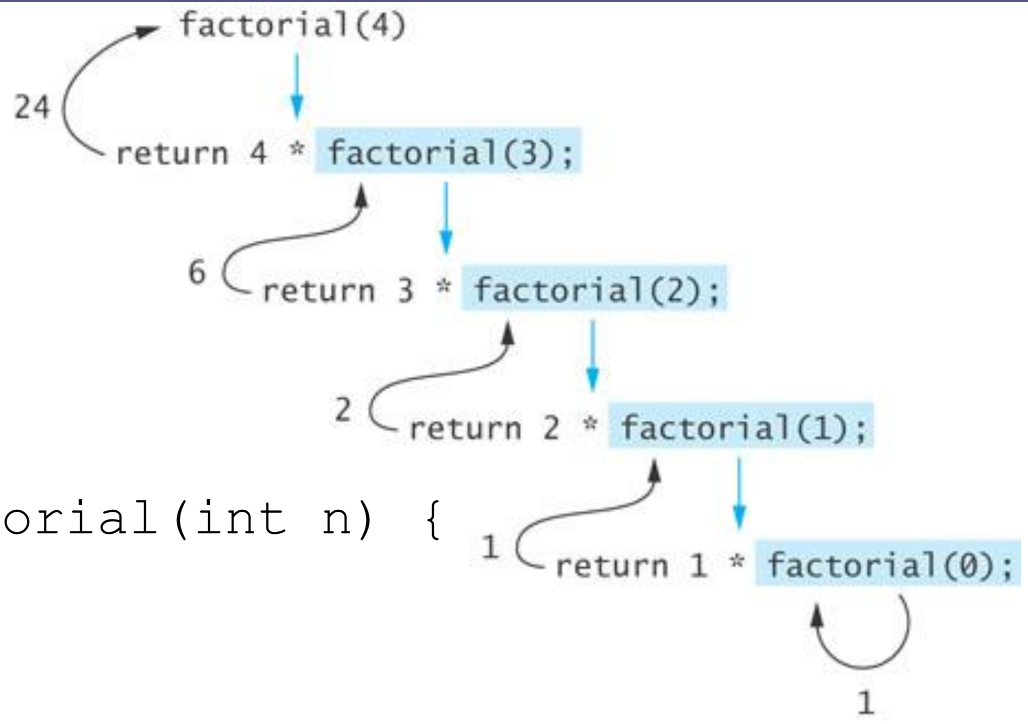
```
if n equals 0
    n! is 1
else
    n! = n x (n – 1)!
```

- The last step can be implemented as:

```
return n * factorial(n - 1);
```



Factorial of *n*: *n*! (cont.)



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

Infinite Recursion and Stack Overflow

- If you call method `factorial` with a negative argument, the recursion will not terminate because `n` will never equal `0`
- If a program does not terminate, it will eventually throw the `StackOverflowError` exception
- Make sure your recursive methods are constructed so that a stopping case is always reached
- In the `factorial` method, you could throw an `IllegalArgumentException` if `n` is negative

Recursive Algorithm for Calculating x^n

Recursive Algorithm for Calculating x^n ($n \geq 0$)

```
if n is 0
The result is 1
else
The result is $x \times x^{n-1}$
```

```
/** Recursive power method (in RecursiveMethods.java).
    pre: n >= 0
    @param x The number being raised to a power
    @param n The exponent
    @return x raised to the power n
*/
public static double power(double x, int n) {
    if (n == 0)
 return 1;
    else
 return x * power(x, n - 1);
}
```

Recursive Algorithm for Calculating gcd

- The greatest common divisor (gcd) of two numbers is the largest integer that divides both numbers
- The gcd of 20 and 15 is 5
- The gcd of 36 and 24 is 12
- The gcd of 38 and 18 is 2
- The gcd of 17 and 97 is 1

Recursive Algorithm for Calculating gcd (cont.)

- Given 2 positive integers m and n (m > n)

```
if n is a divisor of m
    gcd(m, n) = n
else
    gcd (m, n) = gcd (n, m % n)
```

Recursive Algorithm for Calculating **gcd** (cont.)

```
/** Recursive gcd method (in RecursiveMethods.java).
    pre: m > 0 and n > 0
    @param m The larger number
    @param n The smaller number
    @return Greatest common divisor of m and n
*/
public static double gcd(int m, int n) {
    if (m % n == 0)
 return n;
    else
 return gcd(n, m % n);
}
```

Recursion Versus Iteration

- There are similarities between recursion and iteration
- In iteration, a loop repetition condition determines whether to repeat the loop body or exit from the loop
- In recursion, the condition usually tests for a base case
- You can always write an iterative solution to a problem that is solvable by recursion
- A recursive algorithm may be simpler than an iterative algorithm and thus easier to write, code, debug, and read

Iterative `factorial` Method

```
/** Iterative factorial method.
    pre: n >= 0
    @param n The integer whose factorial is being computed
    @return n!
*/
public static int factorialIter(int n) {
    int result = 1;
    for (int k = 1; k <= n; k++)
 result = result * k;
    return result;
}
```

Efficiency of Recursion

- Recursive methods often have slower execution times relative to their iterative counterparts
- The overhead for loop repetition is smaller than the overhead for a method call and return
- If it is easier to conceptualize an algorithm using recursion, then you should code it as a recursive method
- The reduction in efficiency usually does not outweigh the advantage of readable code that is easy to debug

Fibonacci Numbers

- Fibonacci numbers were used to model the growth of a rabbit colony

  $fib_1 = 1$

  $fib_2 = 1$

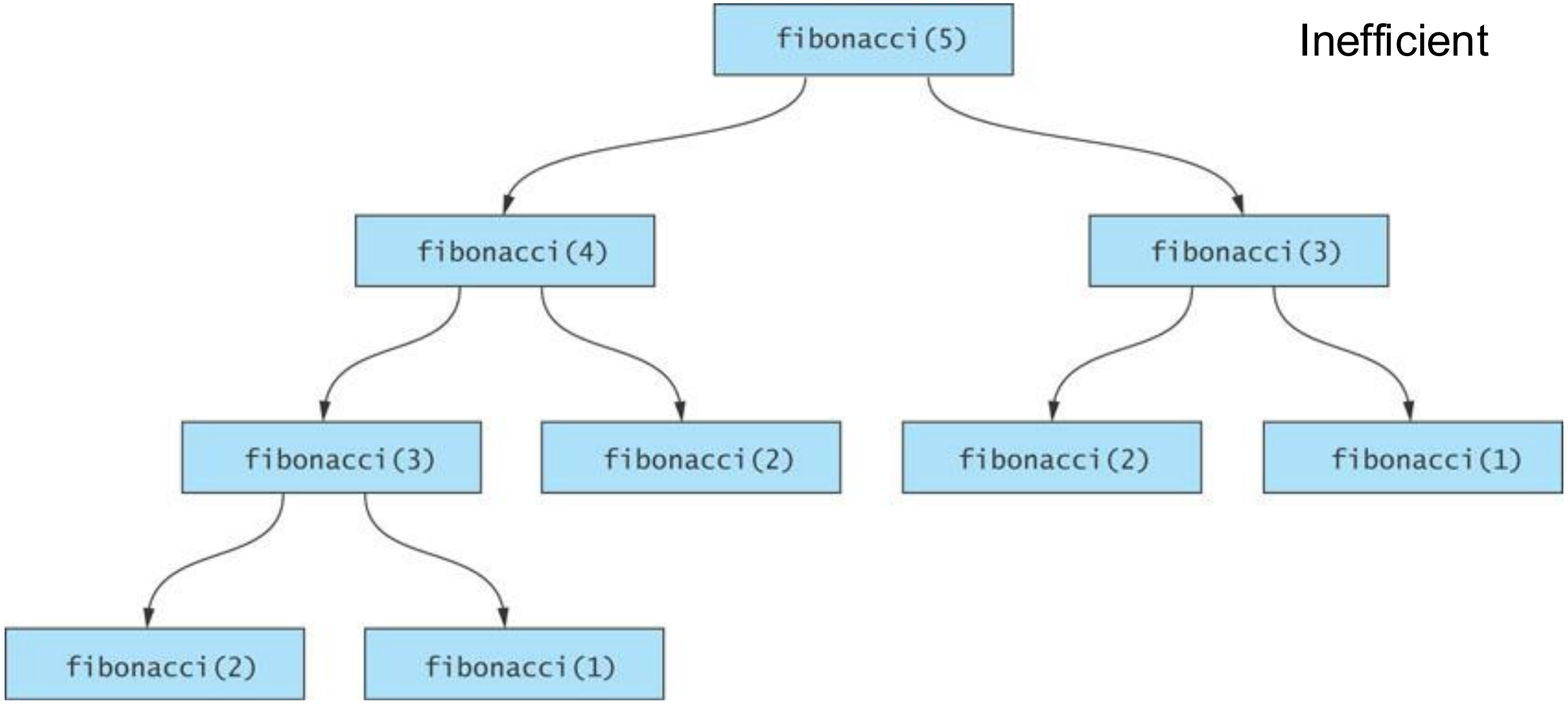
  $fib_n = fib_{n-1} + fib_{n-2}$

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

An Exponential Recursive `fibonacci` Method

```java
/** Recursive method to calculate Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 1
    @param n The position of the Fibonacci number being calculated
    @return The Fibonacci number
*/
public static int fibonacci(int n) {
    if (n <= 2)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Efficiency of Recursion: Exponential fibonacci



Inefficient

An O(n) Recursive `fibonacci` Method

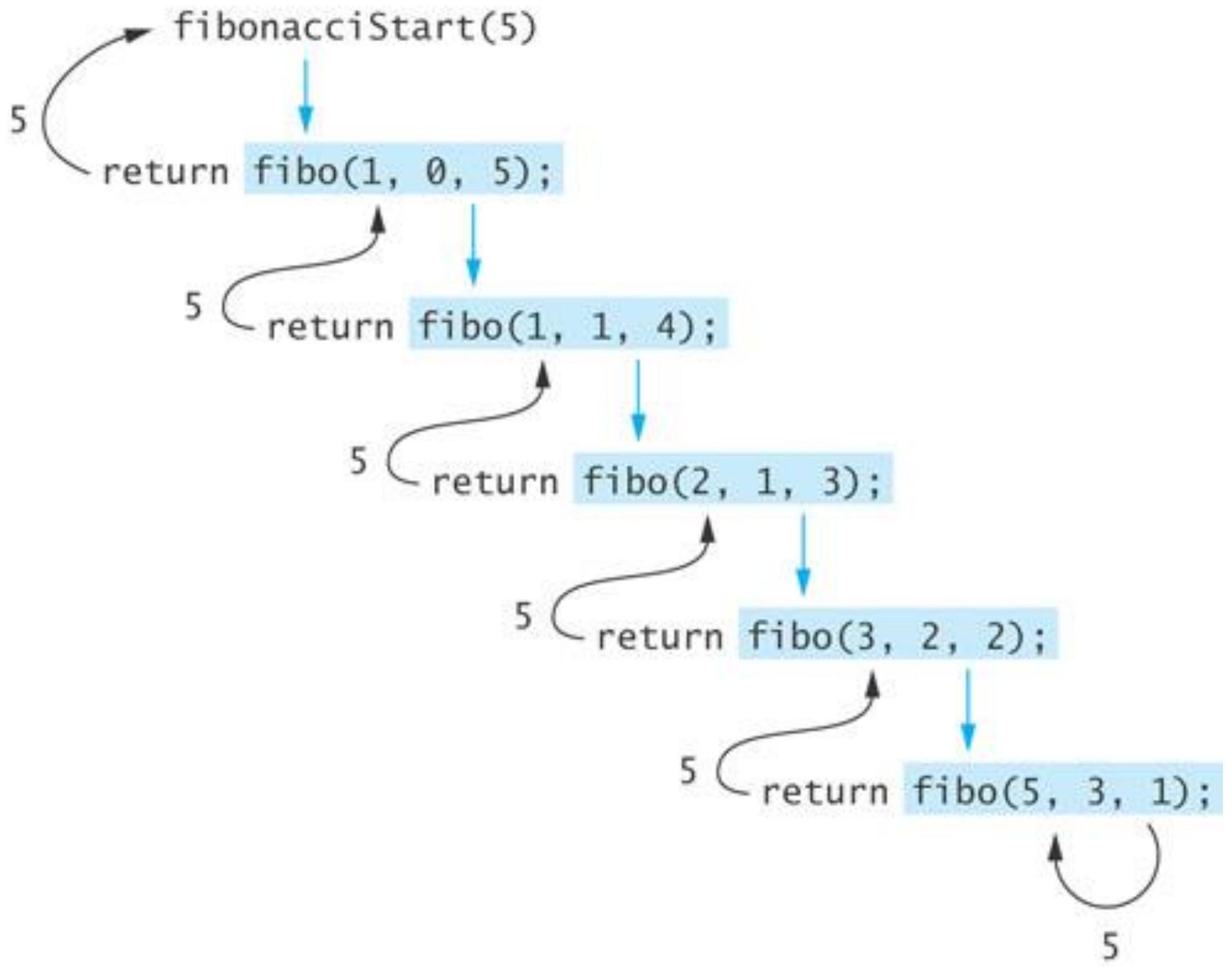
```
/** Recursive O(n) method to calculate Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 1
    @param fibCurrent The current Fibonacci number
    @param fibPrevious The previous Fibonacci number
    @param n The count of Fibonacci numbers left to calculate
    @return The value of the Fibonacci number calculated so far
*/
private static int fibo(int fibCurrent, int fibPrevious, int n) {
    if (n == 1)
        return fibCurrent;
    else
        return fibo(fibCurrent + fibPrevious, fibCurrent, n - 1);
}
```

An O(n) Recursive `fibonacci` Method (cont.)

- In order to start the method executing, we provide a non-recursive wrapper method:

```
/** Wrapper method for calculating Fibonacci numbers
   (in RecursiveMethods.java).
    pre: n >= 1
    @param n The position of the desired Fibonacci
      number
    @return  The value of the nth Fibonacci number
*/
public static int fibonacciStart(int n) {
     return fibo(1, 0, n);
}
```

Efficiency of Recursion: O(n) `fibonacci`



Efficient

Recursive Array Search

Section 5.3

Recursive Array Search

- Searching an array can be accomplished using recursion
- Simplest way to search is a linear search
  - Examine one element at a time starting with the first element and ending with the last
  - On average, $(n + 1)/2$ elements are examined to find the target in a linear search
  - If the target is not in the list, n elements are examined
- A linear search is $O(n)$

Recursive Array Search (cont.)

- Base cases for recursive search:
  - Empty array, target can not be found; result is -1
  - First element of the array being searched = target; result is the subscript of first element
- The recursive step searches the rest of the array, excluding the first element

Algorithm for Recursive Linear Array Search

Algorithm for Recursive Linear Array Search

```
if the array is empty
    the result is –1
else if the first element matches the target
    the result is the subscript of the first element
else
    search the array excluding the first element and return the result
```

Implementation of Recursive Linear Search

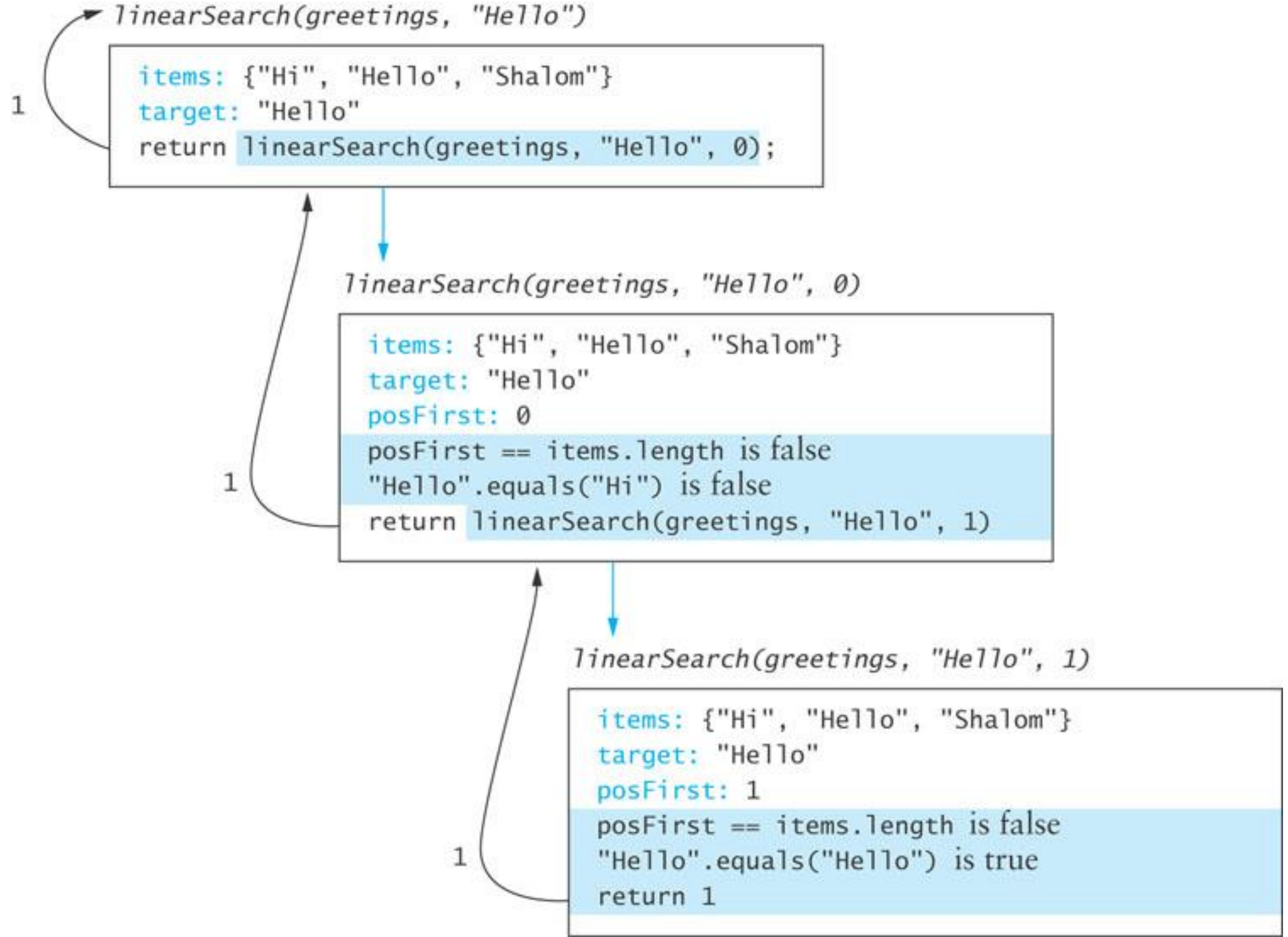
```java
/** Recursive linear search method (in RecursiveMethods.java).
    @param items The array being searched
    @param target The item being searched for
    @param posFirst The position of the current first element
    @return The subscript of target if found; otherwise -1
*/
private static int linearSearch(Object[] items,
                                Object target, int posFirst) {
    if (posFirst == items.length)
        return -1;
    else if (target.equals(items[posFirst]))
        return posFirst;
    else
        return linearSearch(items, target, posFirst + 1);
}
```

Implementation of Recursive Linear Search (cont.)

- A non-recursive wrapper method:

```
/** Wrapper for recursive linear search method
    @param items The array being searched
    @param target The object being searched for
    @return  The subscript of target if found;
             otherwise -1
*/
public static int linearSearch(Object[] items, Object target)
  {
     return linearSearch(items, target, 0);
}
```

Implementation of Recursive Linear Search (cont.)


linearSearch(greetings, "Hello")
items: {"Hi", "Hello", "Shalom"}
target: "Hello"
return linearSearch(greetings, "Hello", 0);
1
linearSearch(greetings, "Hello", 0)
items: {"Hi", "Hello", "Shalom"}
target: "Hello"
posFirst: 0
posFirst == items.length is false
"Hello".equals("Hi") is false
return linearSearch(greetings, "Hello", 1)
1
linearSearch(greetings, "Hello", 1)
items: {"Hi", "Hello", "Shalom"}
target: "Hello"
posFirst: 1
posFirst == items.length is false
"Hello".equals("Hello") is true
return 1
1

Design of a Binary Search Algorithm

- A binary search can be performed only on an array that has been sorted
- Base cases
  - The array is empty
  - Element being examined matches the target
- Rather than looking at the first element, a binary search compares the middle element for a match with the target
- If the middle element does not match the target, a binary search excludes the half of the array within which the target cannot lie

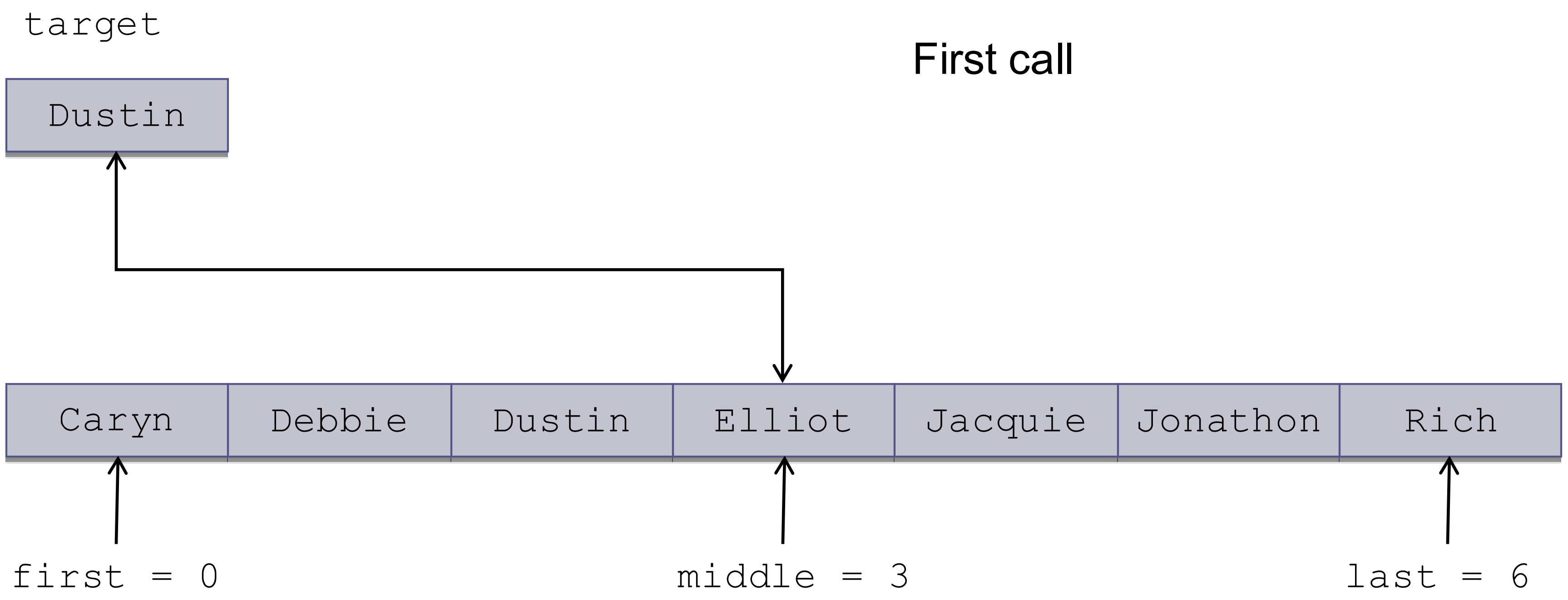
Design of a Binary Search Algorithm (cont.)

Binary Search Algorithm

```
if the array is empty
    return –1 as the search result
else if the middle element matches the target
    return the subscript of the middle element as the result
else if the target is less than the middle element
    recursively search the array elements before the middle element
    and return the result
else
    recursively search the array elements after the middle element and
    return the result
```

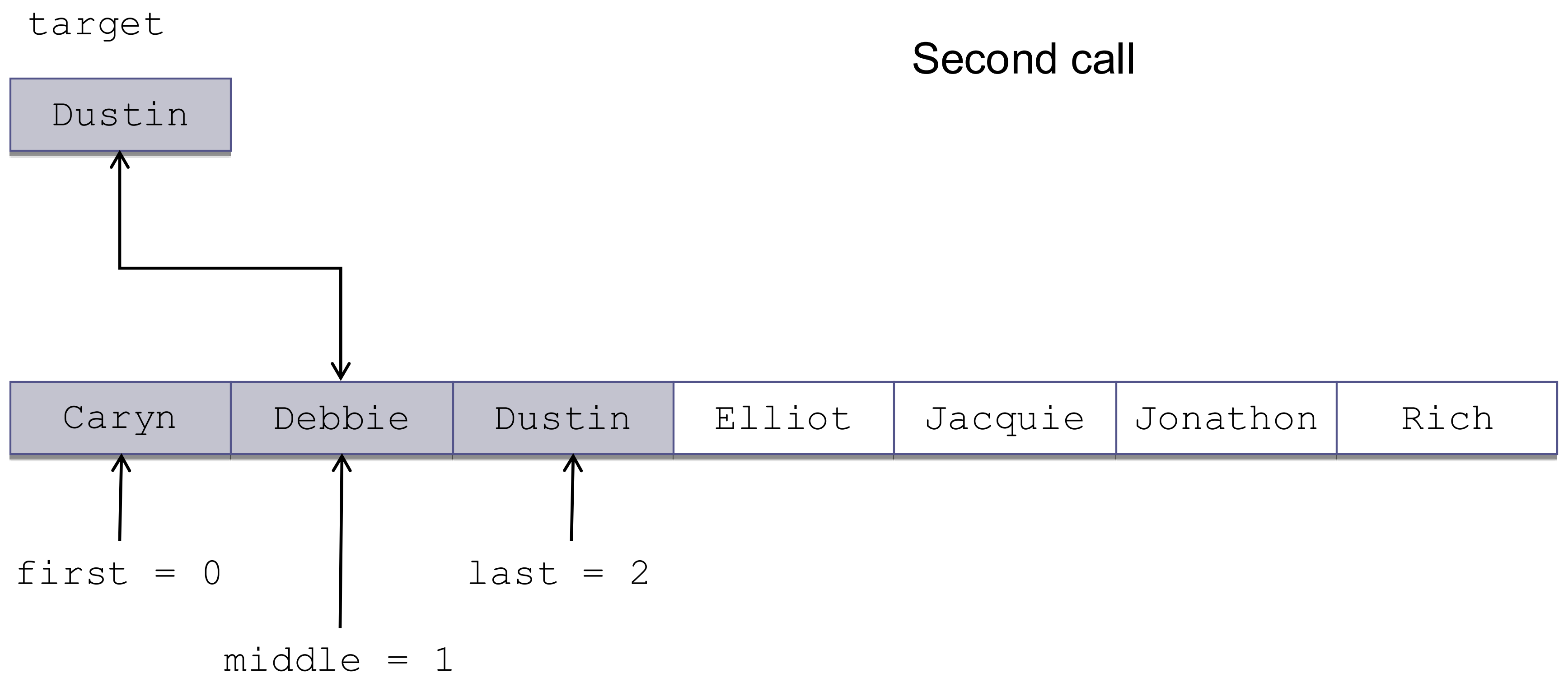
Binary Search Algorithm

First call



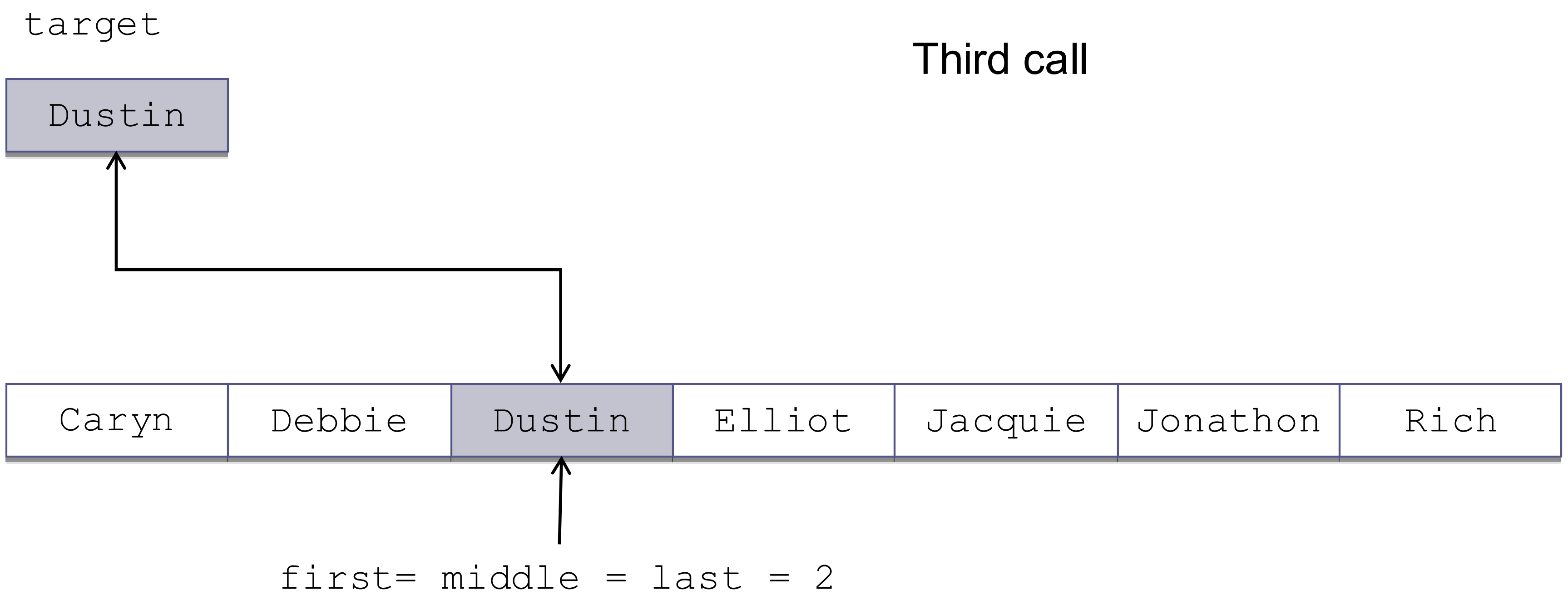
Binary Search Algorithm (cont.)

Second call



Binary Search Algorithm (cont.)

Third call



Efficiency of Binary Search

- At each recursive call we eliminate half the array elements from consideration, making a binary search O(log *n*)
- An array of 16 would search arrays of length 16, 8, 4, 2, and 1: 5 probes in the worst case
  - $16 = 2^4$
  - $5 = \log_2 16 + 1$
- A doubled array size would require only 6 probes in the worst case
  - $32 = 2^5$
  - $6 = \log_2 32 + 1$
- An array with 32,768 elements requires only 16 probes! ($\log_2 32768 = 15$)

`Comparable` Interface

- Classes that implement the `Comparable` interface must define a `compareTo` method
- Method call `obj1.compareTo(obj2)` returns an integer with the following values
  - negative if `obj1 < obj2`
  - zero if `obj1 == obj2`
  - positive if `obj1 > obj2`
- Implementing the `Comparable` interface is an efficient way to compare objects during a search

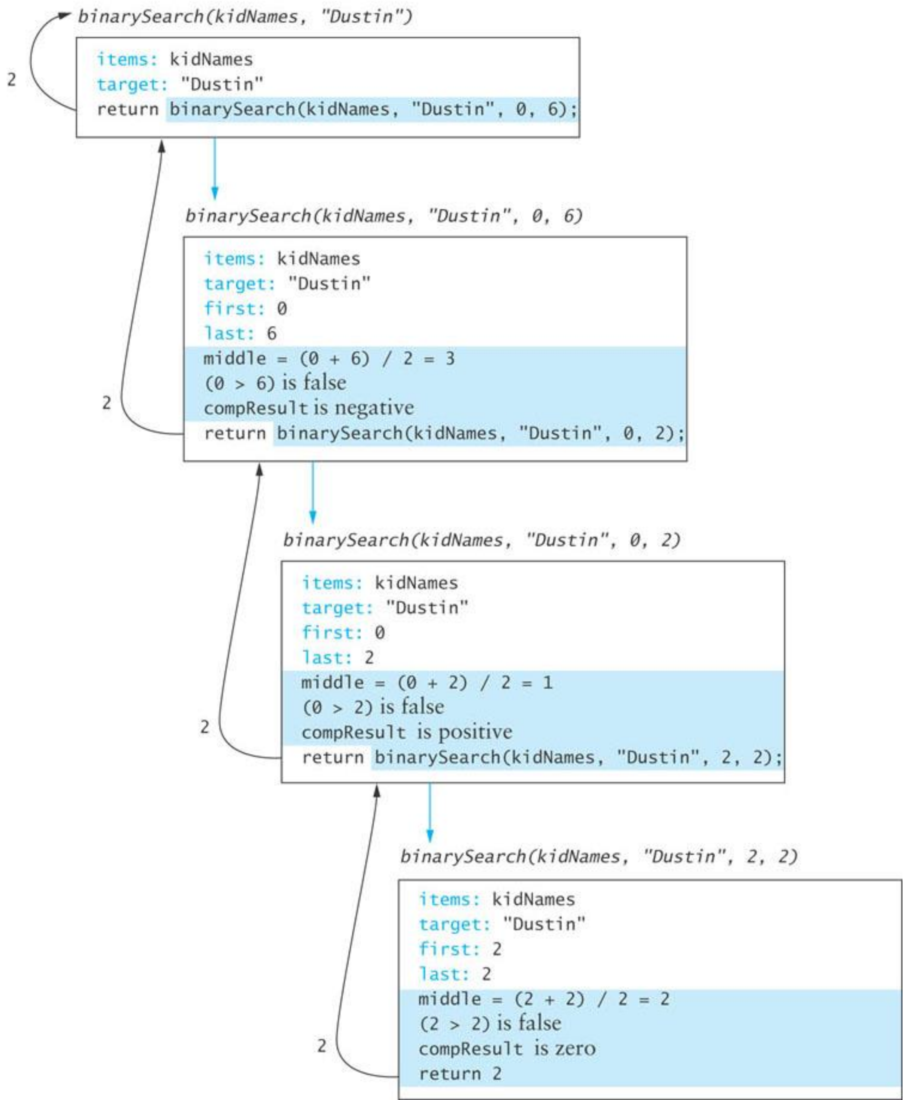
Implementation of a Binary Search Algorithm

```
/** Recursive binary search method (in RecursiveMethods.java).
    @param items The array being searched
    @param target The object being searched for
    @param first The subscript of the first element
    @param last The subscript of the last element
    @return The subscript of target if found; otherwise -1.
*/
private static int binarySearch(Object[] items, Comparable target,
                                int first, int last) {
    if (first > last)
        return -1;     // Base case for unsuccessful search.
    else {
        int middle = (first + last) / 2;  // Next probe index.
        int compResult = target.compareTo(items[middle]);
        if (compResult == 0)
            return middle;   // Base case for successful search.
        else if (compResult < 0)
            return binarySearch(items, target, first, middle - 1);
        else
            return binarySearch(items, target, middle + 1, last);
    }
}
```

Implementation of a Binary Search Algorithm (cont.)

```
/** Wrapper for recursive binary search method (in RecursiveMethods.java).
    @param items The array being searched
    @param target The object being searched for
    @return The subscript of target if found; otherwise -1.
*/
public static int binarySearch(Object[] items, Comparable target) {
    return binarySearch(items, target, 0, items.length - 1);
}
```

Trace of Binary Search



Testing Binary Search

- You should test arrays with
  - an even number of elements
  - an odd number of elements
  - duplicate elements
- Test each array for the following cases:
  - the target is the element at each position of the array, starting with the first position and ending with the last position
  - the target is less than the smallest array element
  - the target is greater than the largest array element
  - the target is a value between each pair of items in the array

Method `Arrays.binarySearch`

- Java API class `Arrays` contains a `binarySearch` method
  - Called with sorted arrays of primitive types or with sorted arrays of objects
  - If the objects in the array are not mutually comparable or if the array is not sorted, the results are undefined
  - If there are multiple copies of the target value in the array, there is no guarantee which one will be found
  - Throws `ClassCastException` if the target is not comparable to the array elements

Recursive Data Structures

Section 5.4

Recursive Data Structures

- Computer scientists often encounter data structures that are defined recursively – each with another version of itself as a component
- Linked lists and trees (Chapter 6) can be defined as recursive data structures
- Recursive methods provide a natural mechanism for processing recursive data structures
- The first language developed for artificial intelligence research was a recursive language called LISP

Recursive Definition of a Linked List

- A linked list is a collection of nodes such that each node references another linked list consisting of the nodes that follow it in the list
- The last node references an empty list
- A linked list is empty, or it contains a node, called the list head, that stores data and a reference to a linked list

Class `LinkedListRec`

- We define a class `LinkedListRec<E>` that implements several list operations using recursive methods

```
public class LinkedListRec<E> {
  private Node<E> head;

  // inner class Node<E> here
  // (from chapter 2)
}
```

Recursive `size` Method

```
/** Finds the size of a list.
    @param head The head of the current list
    @return The size of the current list
*/
private int size(Node<E> head) {
    if (head == null)
        return 0;
    else
        return 1 + size(head.next);
}

/** Wrapper method for finding the size of a list.
    @return The size of the list
*/
public int size() {
    return size(head);
}
```

Recursive `toString` Method

```
/** Returns the string representation of a list.
    @param head The head of the current list
    @return The state of the current list
*/
private String toString(Node<E> head) {
    if (head == null)
        return "";
    else
        return head.data + "\n" + toString(head.next);
}

/** Wrapper method for returning the string representation of a list.
    @return The string representation of the list
*/
public String toString() {
    return toString(head);
}
```

Recursive `replace` Method

```
/** Replaces all occurrences of oldObj with newObj.
    post: Each occurrence of oldObj has been replaced by newObj.
    @param head The head of the current list
    @param oldObj The object being removed
    @param newObj The object being inserted
*/
private void replace(Node<E> head, E oldObj, E newObj) {
    if (head != null) {
        if (oldObj.equals(head.data))
            head.data = newObj;
        replace(head.next, oldObj, newObj);
    }
}

/*  Wrapper method for replacing oldObj with newObj.
    post: Each occurrence of oldObj has been replaced by newObj.
    @param oldObj The object being removed
    @param newObj The object being inserted
*/
public void replace(E oldObj, E newObj) {
    replace(head, oldObj, newObj);
}
```

Recursive `add` Method

```
/** Adds a new node to the end of a list.
    @param head The head of the current list
    @param data The data for the new node
*/
private void add(Node<E> head, E data) {
    // If the list has just one element, add to it.
    if (head.next == null)
        head.next = new Node<E>(data);
    else
        add(head.next, data);       // Add to rest of list.
}

/** Wrapper method for adding a new node to the end of a list.
    @param data The data for the new node
*/
public void add(E data) {
    if (head == null)
        head = new Node<E>(data);  // List has 1 node.
    else
        add(head, data);
}
```

Recursive `remove` Method

```
/** Removes a node from a list.
    post: The first occurrence of outData is removed.
    @param head The head of the current list
    @param pred The predecessor of the list head
    @param outData The data to be removed
    @return true if the item is removed
            and false otherwise
*/
private boolean remove(Node<E> head, Node<E> pred, E outData) {
    if (head == null)  // Base case - empty list.
        return false;
    else if (head.data.equals(outData)) {  // 2nd base case.
        pred.next = head.next;  // Remove head.
        return true;
    } else
        return remove(head.next, head, outData);
}
```

Recursive `remove` Method (cont.)

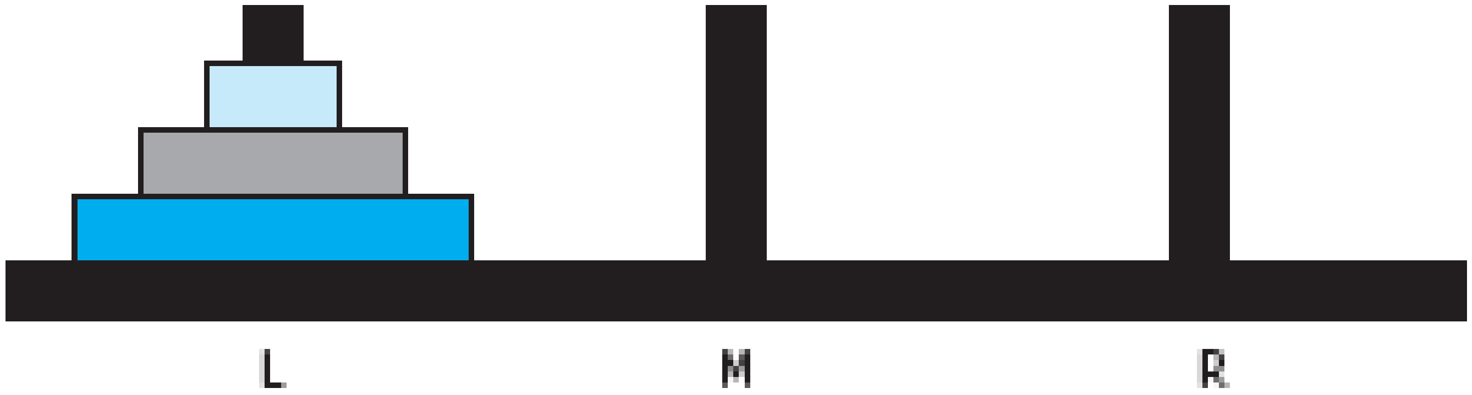
```
/** Wrapper method for removing a node (in LinkedListRec).
    post: The first occurrence of outData is removed.
    @param outData The data to be removed
    @return true if the item is removed,
            and false otherwise
*/
public boolean remove(E outData) {
    if (head == null)
        return false;
    else if (head.data.equals(outData)) {
        head = head.next;
        return true;
    } else
        return remove(head.next, head, outData);
}
```

Problem Solving with Recursion

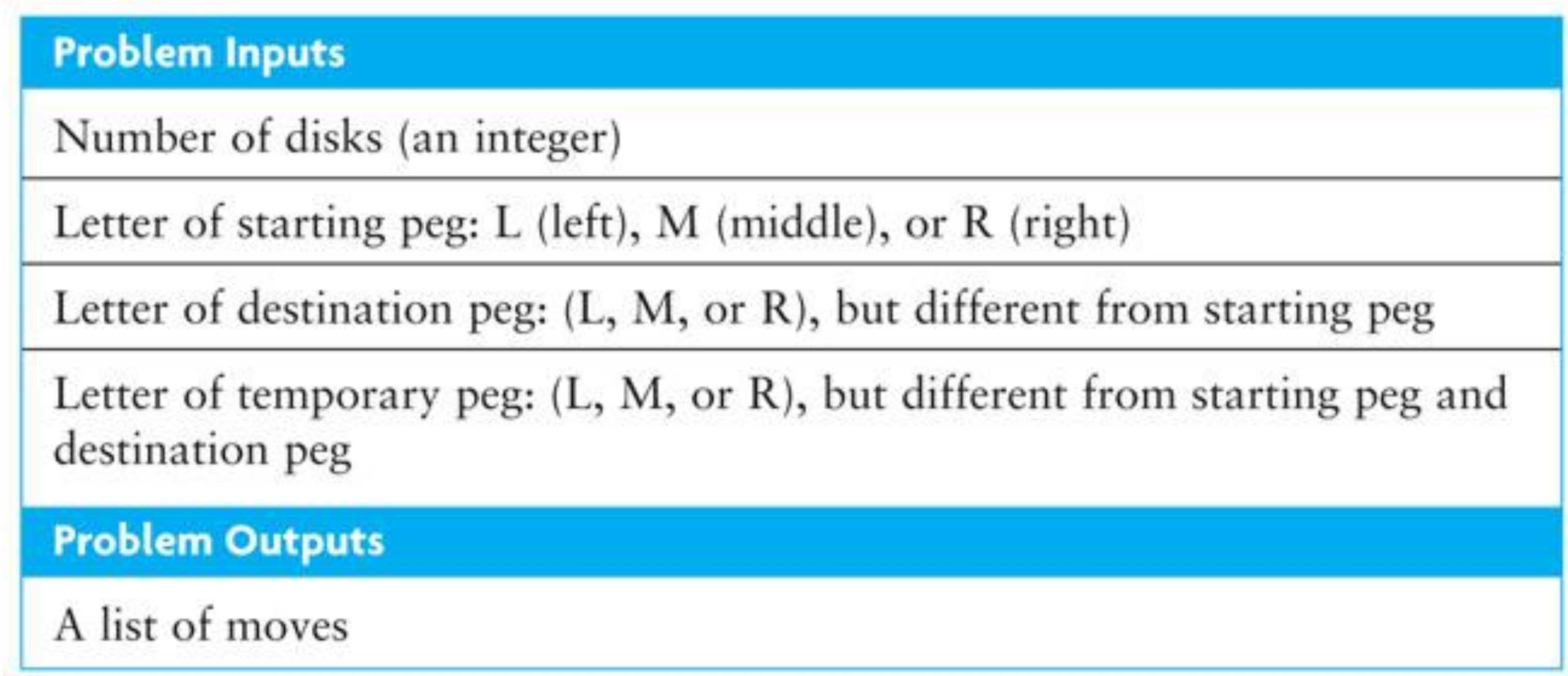
Section 5.5

Simplified Towers of Hanoi

- Move the three disks to a different peg, maintaining their order (largest disk on bottom, smallest on top, etc.)
  - Only the top disk on a peg can be moved to another peg
  - A larger disk cannot be placed on top of a smaller disk



Towers of Hanoi

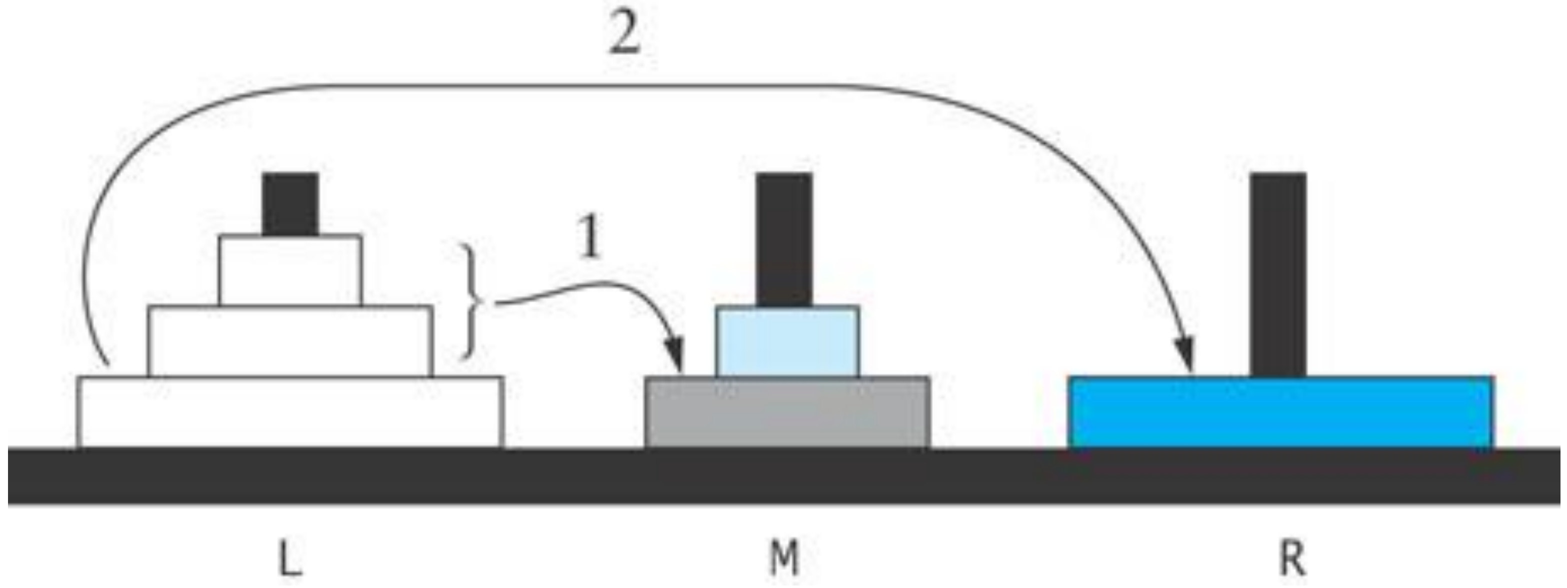


Problem Inputs
Number of disks (an integer)
Letter of starting peg: L (left), M (middle), or R (right)
Letter of destination peg: (L, M, or R), but different from starting peg
Letter of temporary peg: (L, M, or R), but different from starting peg and destination peg
Problem Outputs
A list of moves

Algorithm for Towers of Hanoi

Solution to Three-Disk Problem: Move Three Disks from Peg L to Peg R

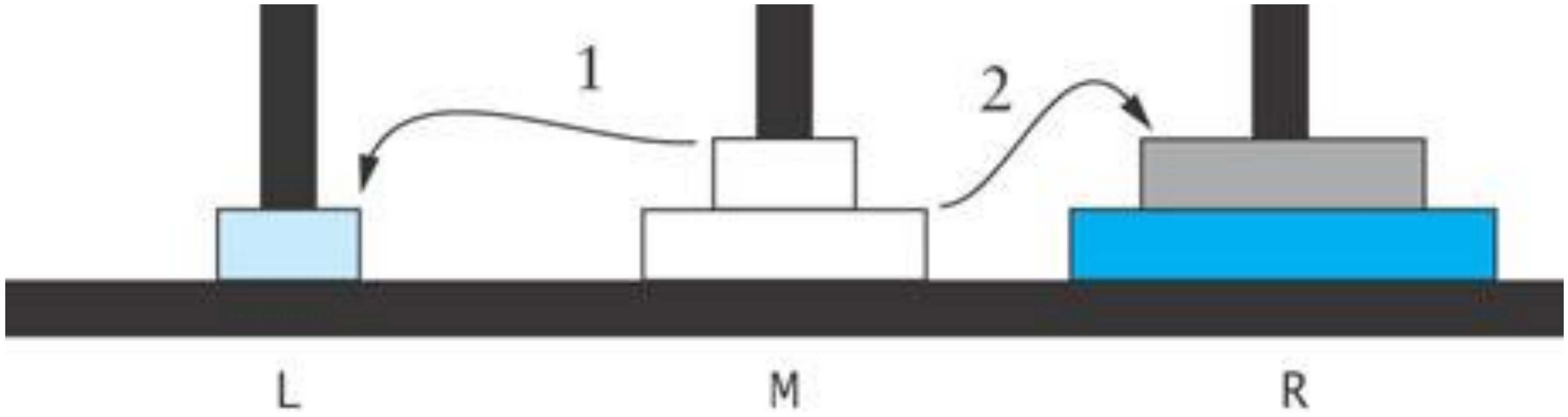
1. Move the top two disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top two disks from peg M to peg R.



Algorithm for Towers of Hanoi (cont.)

Solution to Three-Disk Problem: Move Top Two Disks from Peg M to Peg R

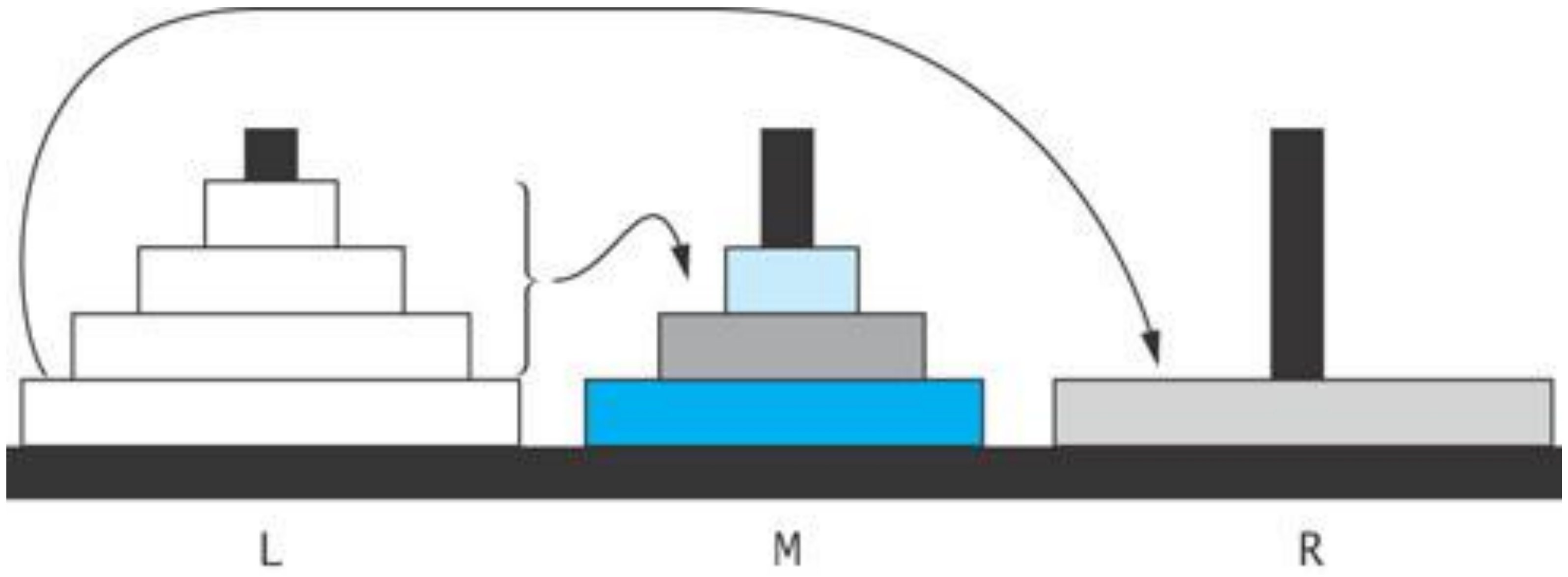
1. Move the top disk from peg M to peg L.
2. Move the bottom disk from peg M to peg R.
3. Move the top disk from peg L to peg R.



Algorithm for Towers of Hanoi (cont.)

Solution to Four-Disk Problem: Move Four Disks from Peg L to Peg R

1. Move the top three disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top three disks from peg M to peg R.

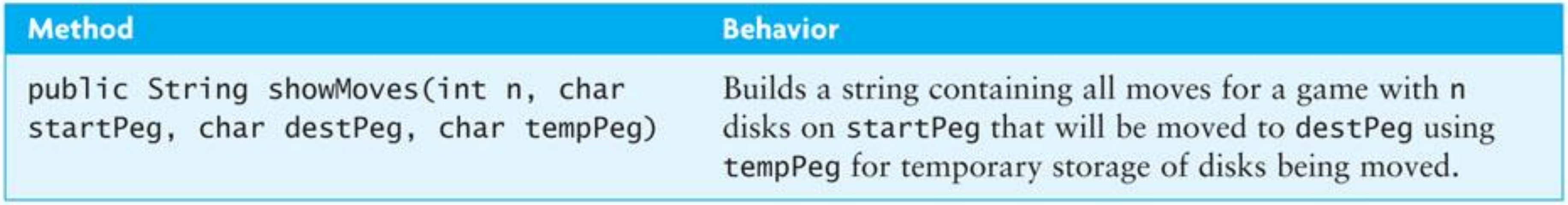


Recursive Algorithm for Towers of Hanoi

Recursive Algorithm for n -Disk Problem: Move *n* Disks from the Starting Peg to the Destination Peg

```
if $n$ is 1
    move disk 1 (the smallest disk) from the starting peg to the destination peg
else
    move the top $n – 1$ disks from the starting peg to the temporary peg (neither starting nor destination peg)
    move disk $n$ (the disk at the bottom) from the starting peg to the destination peg
    move the top $n – 1$ disks from the temporary peg to the destination peg
```

Recursive Algorithm for Towers of Hanoi (cont.)



Method	Behavior
`public String showMoves(int n, char startPeg, char destPeg, char tempPeg)`	Builds a string containing all moves for a game with `n` disks on `startPeg` that will be moved to `destPeg` using `tempPeg` for temporary storage of disks being moved.

Implementation of Recursive Towers of Hanoi

```java
/** Class that solves Towers of Hanoi problem. */
public class TowersOfHanoi {
    /** Recursive method for "moving" disks.
        pre: startPeg, destPeg, tempPeg are different.
        @param n is the number of disks
        @param startPeg is the starting peg
        @param destPeg is the destination peg
        @param tempPeg is the temporary peg
        @return A string with all the required disk moves
    */
    public static String showMoves(int n, char startPeg,
                                   char destPeg, char tempPeg) {
        if (n == 1) {
            return "Move disk 1 from peg " + startPeg +
                   " to peg " + destPeg + "\n";
        } else {  // Recursive step
            return showMoves(n - 1, startPeg, tempPeg, destPeg)
                   + "Move disk " + n + " from peg " + startPeg
                   + " to peg " + destPeg + "\n"
                   + showMoves(n - 1, tempPeg, destPeg, startPeg);
        }
    }
}
```

Counting Cells in a Blob

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

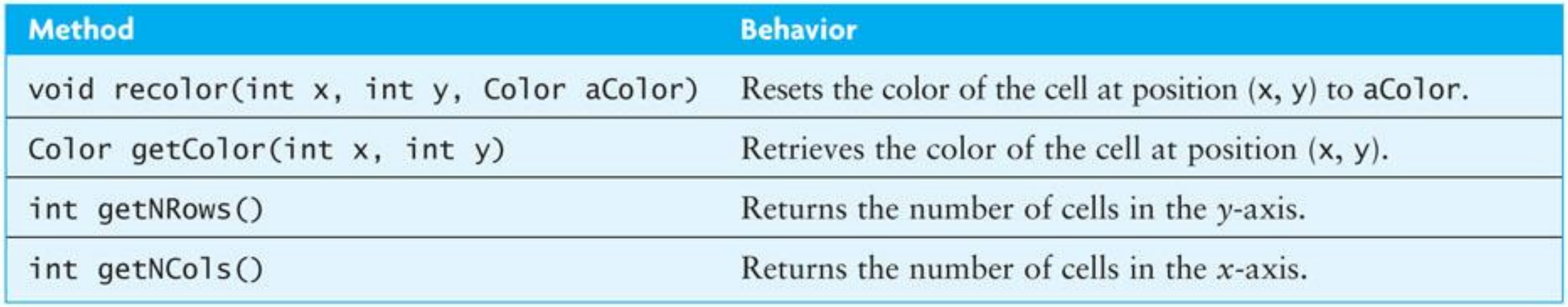
Counting Cells in a Blob – the Problem

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

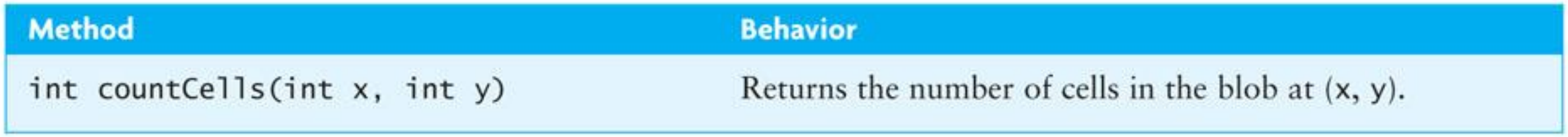
Counting Cells in a Blob - Analysis

- Problem Inputs
  - the two-dimensional grid of cells
  - the coordinates of a cell in a blob
- Problem Outputs
  - the count of cells in the blob

Counting Cells in a Blob - Design



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



Method	Behavior
`int countCells(int x, int y)`	Returns the number of cells in the blob at (x, y).

Counting Cells in a Blob - Design (cont.)

Algorithm for `countCells(x, y)`

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

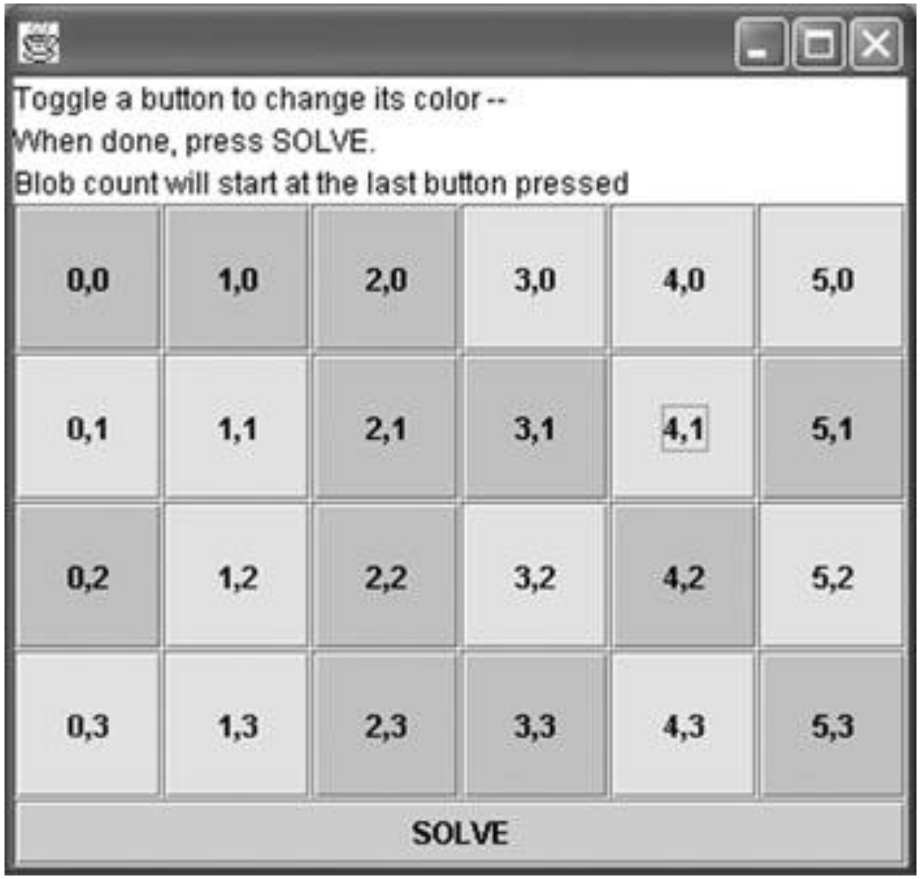
Counting Cells in a Blob - Implementation

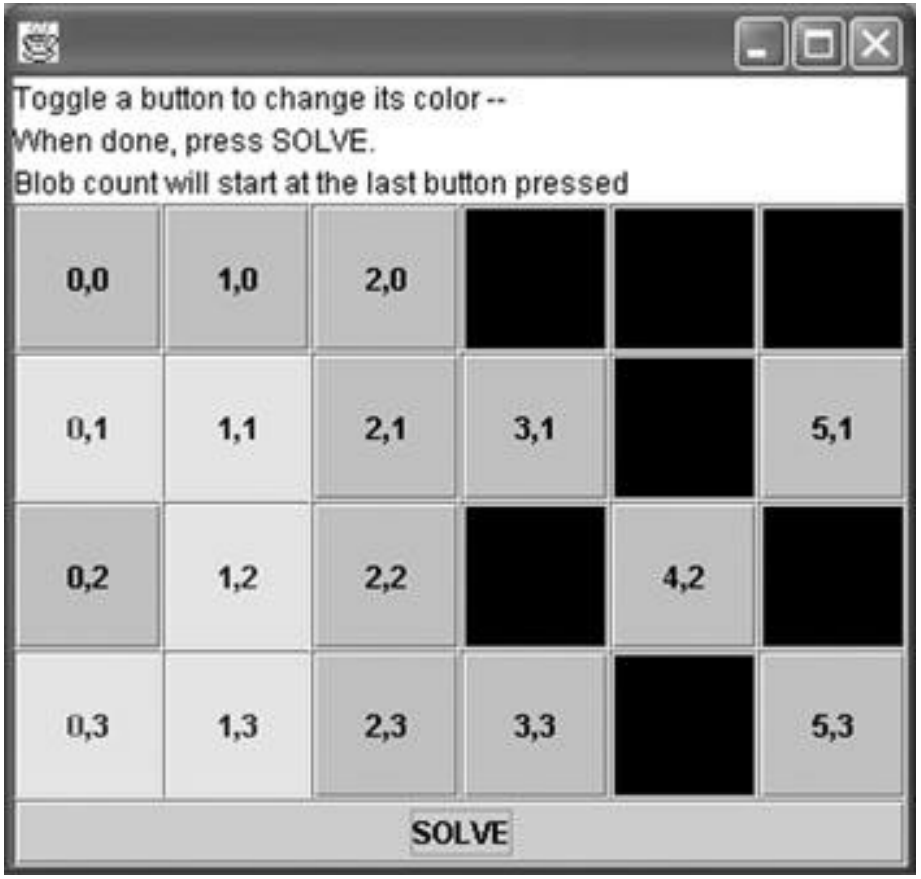
```
import java.awt.*;

/** Class that solves problem of counting abnormal cells. */
public class Blob implements GridColors {

    /** The grid */
    private TwoDimGrid grid;

    /** Constructors */
    public Blob(TwoDimGrid grid) {
        this.grid = grid;
    }
```

Counting Cells in a Blob - Implementation (cont.)

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

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

Counting Cells in a Blob -Testing


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


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

Counting Cells in a Blob -Testing (cont.)

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

Backtracking

Section 5.6

Backtracking

- Backtracking is an approach to implementing a systematic trial and error search for a solution
- An example is finding a path through a maze
- If you are attempting to walk through a maze, you will probably walk down a path as far as you can go
  - Eventually, you will reach your destination or you won't be able to go any farther
  - If you can't go any farther, you will need to consider alternative paths
- Backtracking is a systematic, nonrepetitive approach to trying alternative paths and eliminating them if they don't work

Backtracking (cont.)

- If you never try the same path more than once, you will eventually find a solution path if one exists
- Problems that are solved by backtracking can be described as a set of choices made by some method
- Recursion allows you to implement backtracking in a relatively straightforward manner
  - Each activation frame is used to remember the choice that was made at that particular decision point
- A program that plays chess may involve some kind of backtracking algorithm

Finding a Path through a Maze

- Problem
  - Use backtracking to find a display the path through a maze
  - From each point in a maze you can move to the next cell in a horizontal or vertical direction if the cell is not blocked

Finding a Path through a Maze (cont.)

- Analysis
  - The maze will consist of a grid of colored cells
  - The starting point is at the top left corner (0,0)
  - The exit point is at the bottom right corner `(getNCols() - 1, getNRow -1)`
  - All cells on the path will be `BACKGROUND` color
  - All cells that represent barriers will be `ABNORMAL` color
  - Cells that we have visited will be `TEMPORARY` color
  - If we find a path, all cells on the path will be set to `PATH` color

Recursive Algorithm for Finding Maze Path

Recursive Algorithm for `findMazePath(x, y)`

```
if the current cell is outside the maze
    return false (you are out of bounds)
else if  the current cell is part of the barrier or has been visited already
    return false (you are off the path or in a cycle)
else if  the current cell is the maze exit
    recolor it to the path color and return true (you have successfully
    completed the maze)
else //  Try to find a path from the current path to the exit:
    mark the current cell as on the path by recoloring it to the path color
    for each neighbor of the current cell
        if a path exists from the neighbor to the maze exit
            return true
    // No neighbor of the current cell is on the path
    recolor the current cell to the temporary color (visited) and return
    false
```

Implementation

- Listing 5.4 (`Maze.java`, pages 286-287)

Testing

- Test for a variety of test cases:
  - Mazes that can be solved
  - Mazes that can't be solved
  - A maze with no barrier cells
  - A maze with a single barrier cell at the exit point