

Advanced Programming Techniques in Java



COSI 12B

Recursion (cont.)



Lecture 21



Class Objectives

- More Recursion (Sections 12.1-12.3)
- Backtracking (Section 12.5)



Review: Why learn recursion?

- "cultural experience" - A different way of thinking of problems
- Can solve some kinds of problems better than iteration
- Leads to elegant, simplistic, short code (when used well)
- Many programming languages ("functional" languages such as Scheme, ML, OCaml and Haskell) use recursion exclusively (no loops)



Review: 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



Review: 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



Review: 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 does not outweigh the advantage of readable code that is easy to debug



Review: 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
- A binary search excludes the half of the array within which the target cannot lie



Review: 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



Recursive Data Structures



Recursive Data Structures

- ☐ Computer scientists often encounter data structures that are defined recursively – with another version of itself as a component
- ☐ Linked lists and trees 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, it 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  
  
    //  
  
}
```



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



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





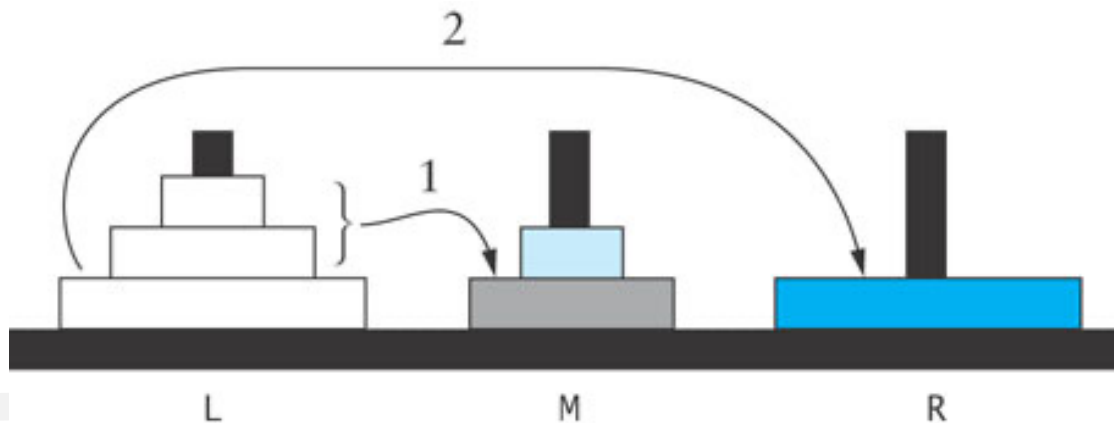
Towers of Hanoi

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

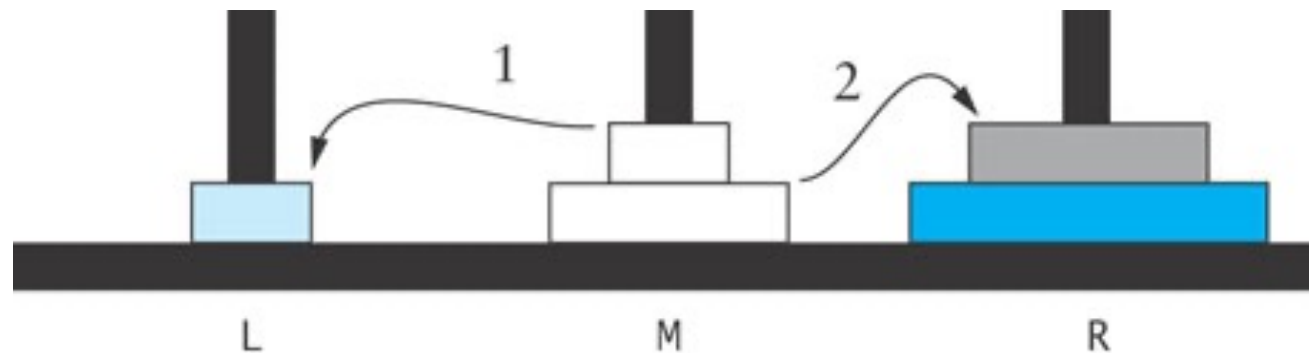
1. Move the top two disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top two disks from peg M to peg R.



Algorithm for Towers of Hanoi (cont.)

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

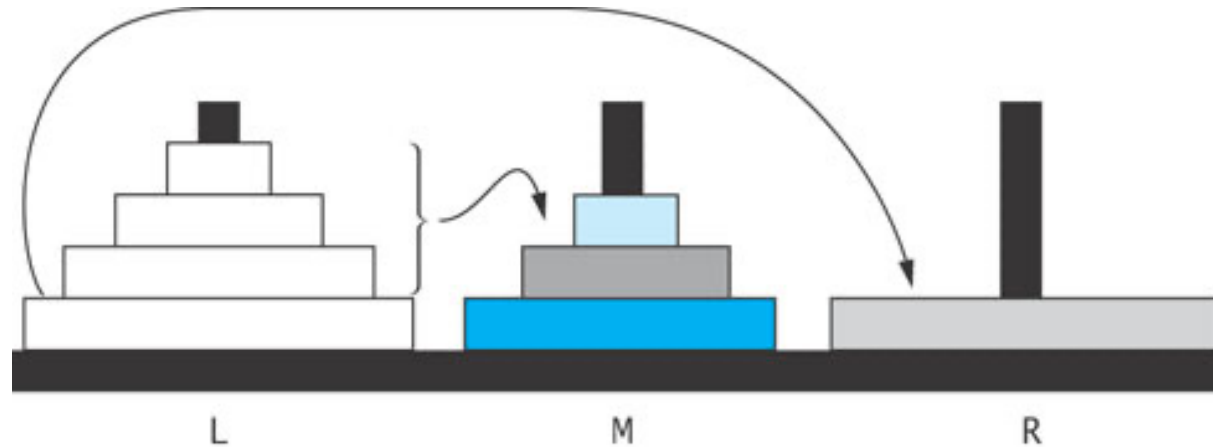
1. Move the top disk from peg M to peg L.
2. Move the bottom disk from peg M to peg R.
3. Move the top disk from peg L to peg R.



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



Implementation of Recursive Towers of Hanoi

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


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



Counting Cells in a Blob - Design (cont.)

Algorithm for `countCells(x, y)`

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


Counting Cells in a Blob - Implementation

```
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

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

SOLVE

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

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

SOLVE



Counting Cells in a Blob -Testing (cont.)

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