



STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

CS 570: Data Structures

Recursion (Part II)

Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu



CHAPTER 5: RECURSION

Part II

Chapter Objectives

3

- ❑ 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

4

- 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

Week 9

- Reading Assignment: Koffman and Wolfgang, Sections 5.4-5.6

Recursive Data Structures

Section 5.4

Recursive Data Structures

7

- ❑ Computer scientists often encounter data structures that are defined recursively – 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

8

- 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

9

- 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

10

```
/** 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

11

```
/** 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

12

```
/** 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

13

```
/** 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

14

```
/** 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.)

15

```
/** 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

17

- 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

18

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

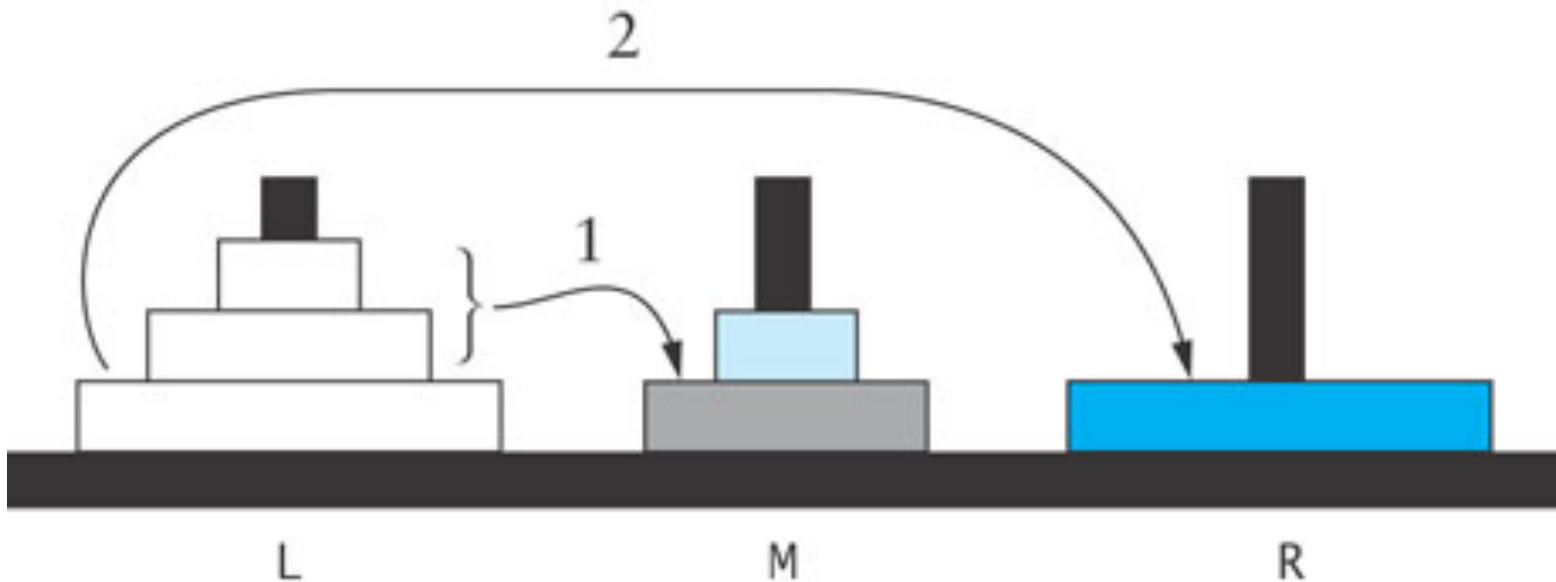
A list of moves

Algorithm for Towers of Hanoi

19

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.

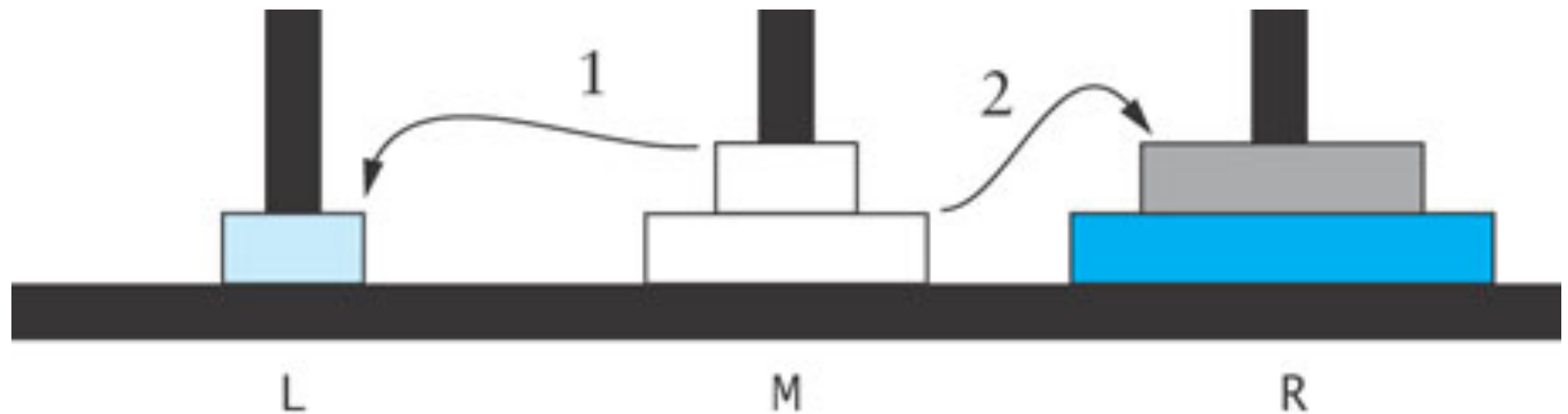


Algorithm for Towers of Hanoi (cont.)

20

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.

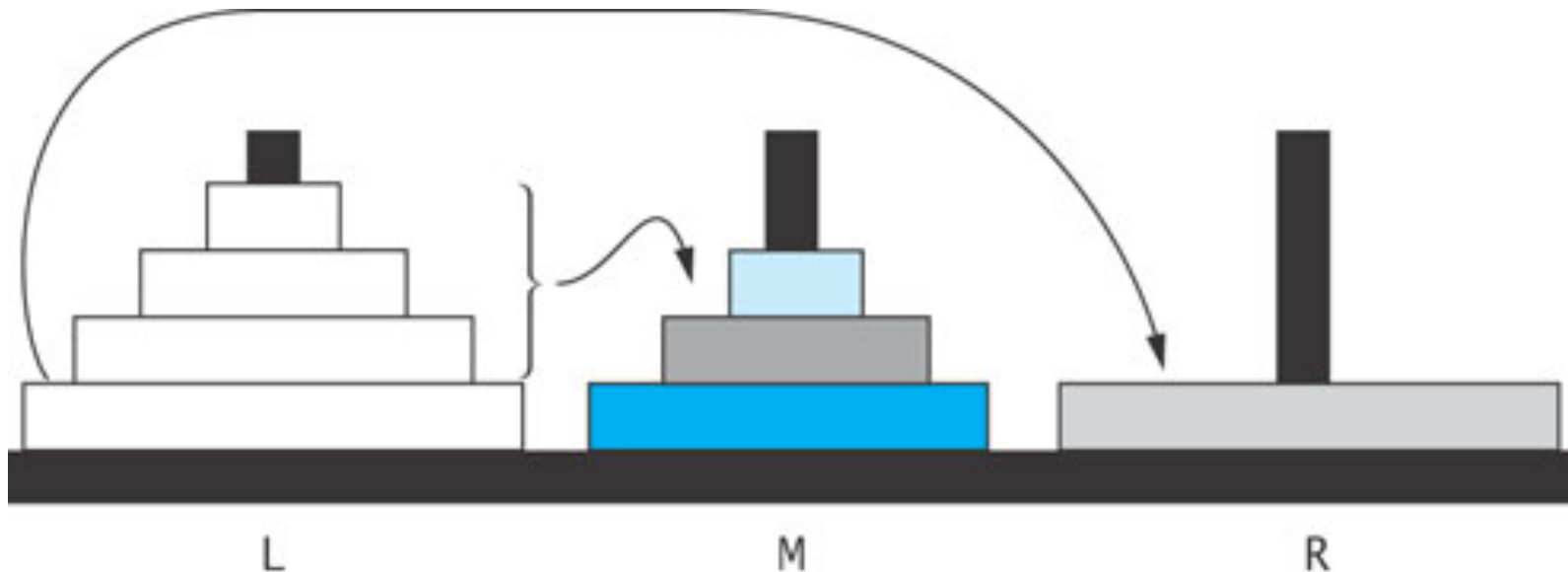


Algorithm for Towers of Hanoi (cont.)

21

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

22

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

23

```
/** 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

24

- 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

25

- 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

26

- 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

27

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.)

28

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

29

```
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.)





30

```
/** 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





31

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.)

32

- 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

34

- 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.)

35

- 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

36

□ Problem

- Use backtracking to find and 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.)

37

□ 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, getNRows() - 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

38

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 already been visited
    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

39

```
import java.awt.*;

/** Class that solves maze problems with
    backtracking.
    *   @author Koffman and Wolfgang
    . */

public class Maze implements GridColors {

    /** The maze */
    private TwoDimGrid maze;

    public Maze(TwoDimGrid m) {
        maze = m;
    }
}
```

Implementation

40

```
/** Wrapper method. */
public boolean findMazePath() {
    return findMazePath(0, 0); // (0, 0) is the start point.
}

/** Attempts to find a path through point (x, y).
    pre: Possible path cells are in BACKGROUND color;
         barrier cells are in ABNORMAL color.
    post: If a path is found, all cells on it are set to the
          PATH color; all cells that were visited but are
          not on the path are in the TEMPORARY color.
    @param x The x-coordinate of current point
    @param y The y-coordinate of current point
    @return If a path through (x, y) is found, true;
            otherwise, false
*/
```


Implementation

41

```
public boolean findMazePath(int x, int y) {
    if (x < 0 || y < 0
        || x >= maze.getNCols() || y >= maze.getNRows())
        return false; // Cell is out of bounds.
    else if (!maze.getColor(x, y).equals(BACKGROUND))
        return false; // Cell is on barrier or dead end.
    else if (x == maze.getNCols() - 1
        && y == maze.getNRows() - 1) {
        maze.recolor(x, y, PATH); // Cell is on path
        return true; // and is maze exit.
    }
```

Implementation

42

```
else { // Recursive case.
    // Attempt to find a path from each neighbor.
    // Tentatively mark cell as on path.
    maze.recolor(x, y, PATH);
    if (findMazePath(x - 1, y)
        || findMazePath(x + 1, y)
        || findMazePath(x, y - 1)
        || findMazePath(x, y + 1)) {
        return true;
    }
    else {
        maze.recolor(x, y, TEMPORARY); // Dead end.
        return false;
    }
}
```

Testing

43

- 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