Recursion

Based on Chapter 7 of
Koffmann and Wolfgang

Famous Quotations

- To err is human, to forgive divine.
  - Alexander Pope, *An Essay on Criticism,* English poet and satirist (1688 - 1744)

- To iterate is human, to recurse, divine.
  - L. Peter Deutsch, computer scientist, or ....
  - Robert Heller, computer scientist, or ....
  - unknown ....

Chapter Outline

- *Thinking recursively*
- *Tracing execution* of a recursive method
- *Writing recursive* algorithms
  - Methods for searching arrays
- Recursive *data structures*
  - Recursive methods for a LinkedList class
- *Solving* the Towers of Hanoi problem with recursion
- *Processing* 2-D images with recursion
- *Backtracking* to solve searchproblems, as in mazes

Recursive Thinking

- ***Recursion*** is:
  - A *problem-solving* ***approach***, that can ...
  - Generate *simple solutions* to ...
  - *Certain kinds* of problems that ...
  - Would be *difficult to solve in other ways*
- Recursion *splits a problem*:
  - Into one or more *simpler versions of* ***itself***

Recursive Thinking: An Example

Strategy for processing nested dolls:

1. if there is only one doll
2. do what it needed for it

else

3. do what is needed for the outer doll
4. Process the inner nest in the same way

FIGURE 7.1
A Set of Nested Wooden Figures

Recursive Thinking: Another Example

Strategy for searching a sorted array:

1. if the array is empty
2. return -1 as the search result (not present)
3. else if the middle element == target
4. return subscript of the middle element
5. else if target < middle element
6. recursively search elements before middle
7. else
8. recursively search elements after the middle

Recursive Thinking: The General Approach

1. if problem is "*small enough*"
2. solve it *directly*
3. else
4. break into one or more *smaller subproblems*
5. solve each subproblem *recursively*
6. *combine* results into solution to whole problem

Requirements for Recursive Solution

- At least one "*small*" case that you can solve directly
- A way of *breaking* a larger problem down into:
  - One or more *smaller* subproblems
  - Each of the *same kind* as the original
- A way of *combining* subproblem results into an overall solution to the larger problem

General Recursive Design Strategy

- Identify the *base case(s)* (for direct solution)
- Devise a problem *splitting strategy*
  - Subproblems must be smaller
  - Subproblems must work towards a base case
- Devise a solution *combining strategy*

Recursive Design Example

Recursive algorithm for finding length of a string:

1. if string is empty (no characters)
2. return 0 ← *base case*
3. else ← *recursive case*
4. compute length of string without first character
5. return 1 + that length

Note: Not best technique for this problem; illustrates the approach.

Recursive Design Example: Code

Recursive algorithm for finding length of a string:

```
public static int length (String str) {
  if (str == null ||
      str.equals(“”))
    return 0;
  else
    return length(str.substring(1)) + 1;
}
```

Recursive Design Example: **`printChars`**

Recursive algorithm for printing a 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 Design Example: **`printChars2`**

Recursive algorithm for printing a string?

```
public static void printChars2
    (String str) {
  if (str == null ||
      str.equals(“”))
    return;
  else
    printChars2(str.substring(1));
    System.out.println(str.charAt(0));
}
```

Recursive Design Example: `mystery`

What does this do?

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

Proving a Recursive Method Correct

Recall ***Proof by Induction:***

1. Prove the theorem for the *base case(s):* ***n=0***
2. Show that:
   - ***If*** the theorem is *assumed true* for ***n***,
   - ***Then*** it *must be true* for ***n+1***

Result: Theorem true for all n ≥ 0.

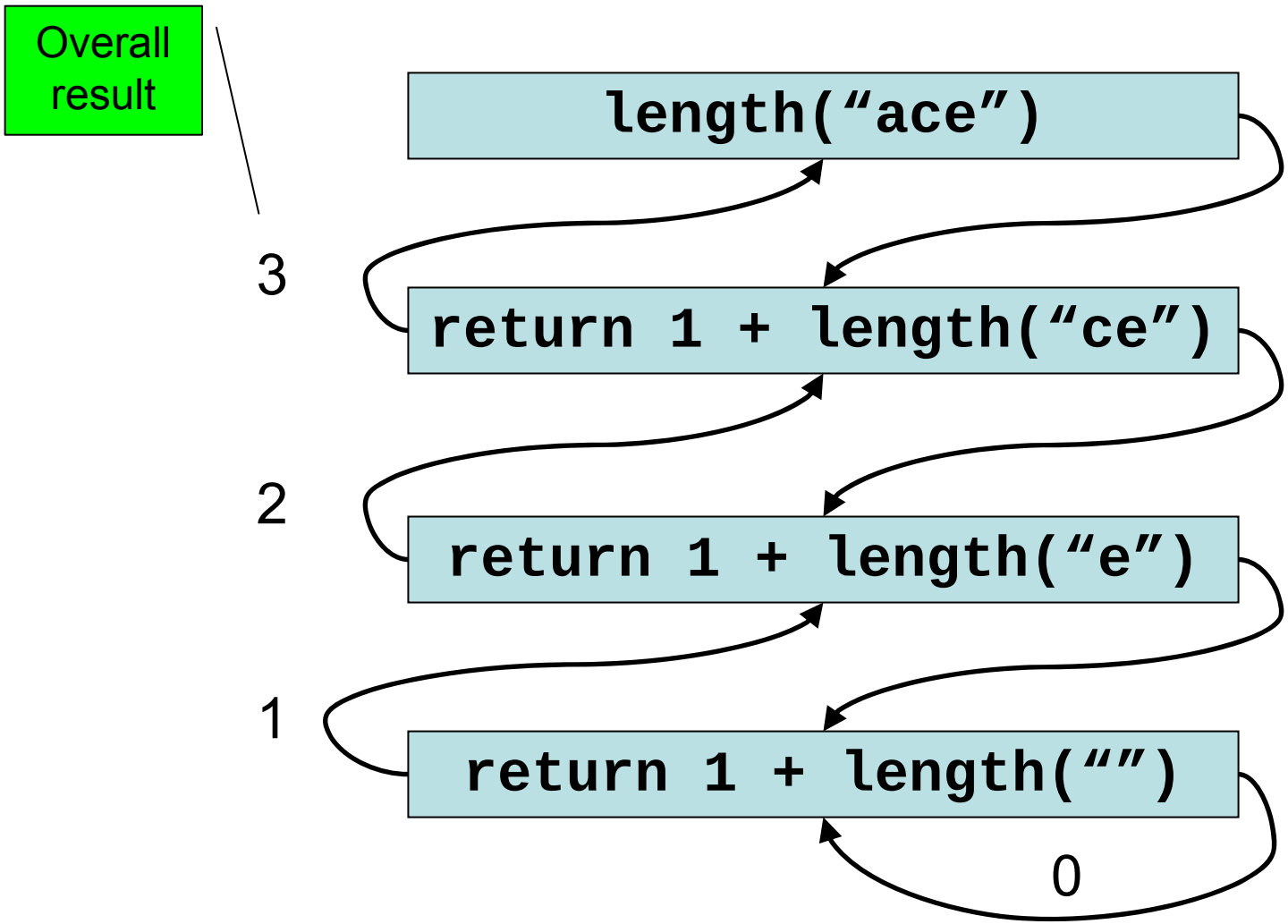
Proving a Recursive Method Correct (2)

Recursive proof is similar to induction:

1. Show *base case* recognized and *solved correctly*
2. Show that
   - ***If*** all *smaller problems* are *solved correctly*,
   - ***Then*** *original problem* is also *solved correctly*

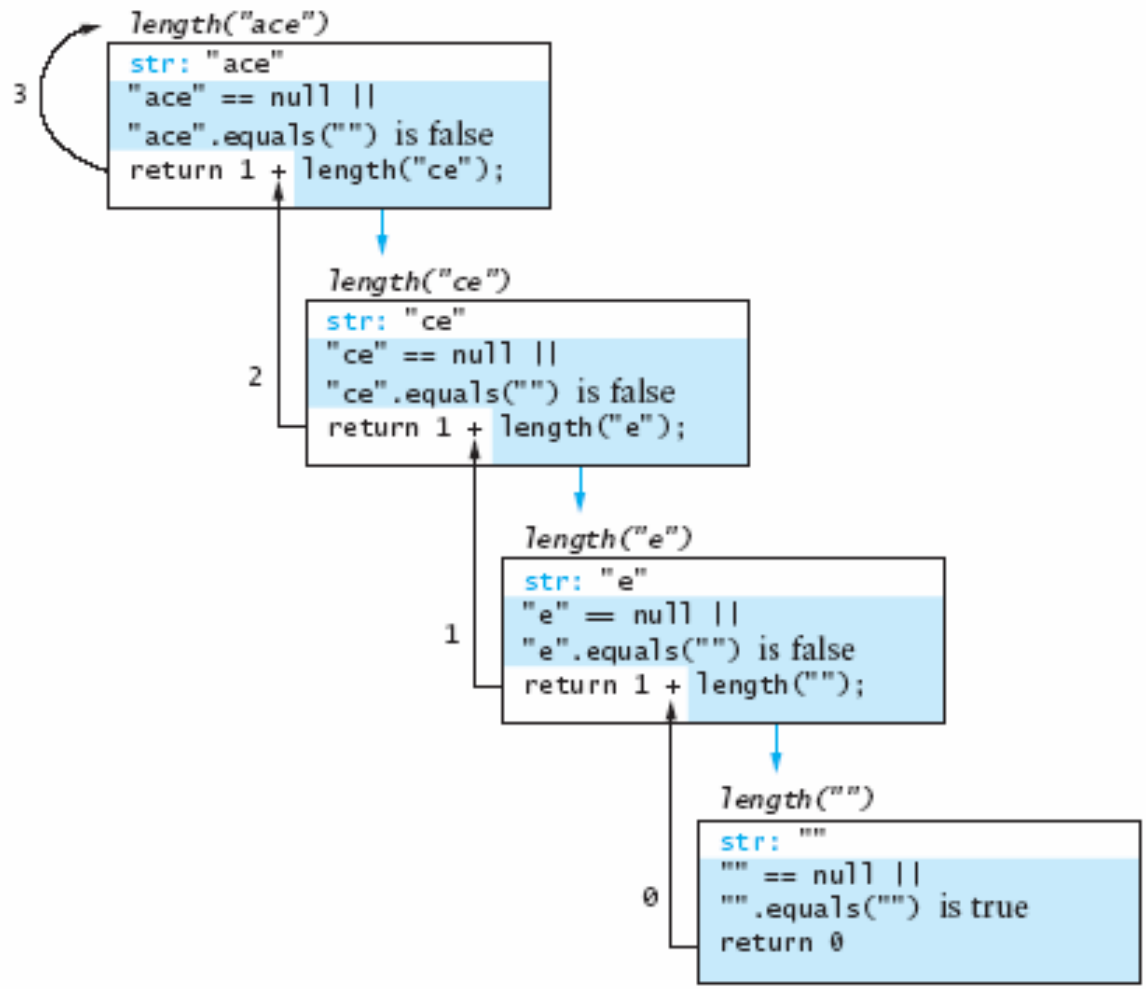
3. Show that each recursive case makes progress towards the base case ⇐ *terminates properly*

Tracing a Recursive Method



Tracing a Recursive Method (2)

FIGURE 7.4
Trace of `length("ace")` Using Activation Frames

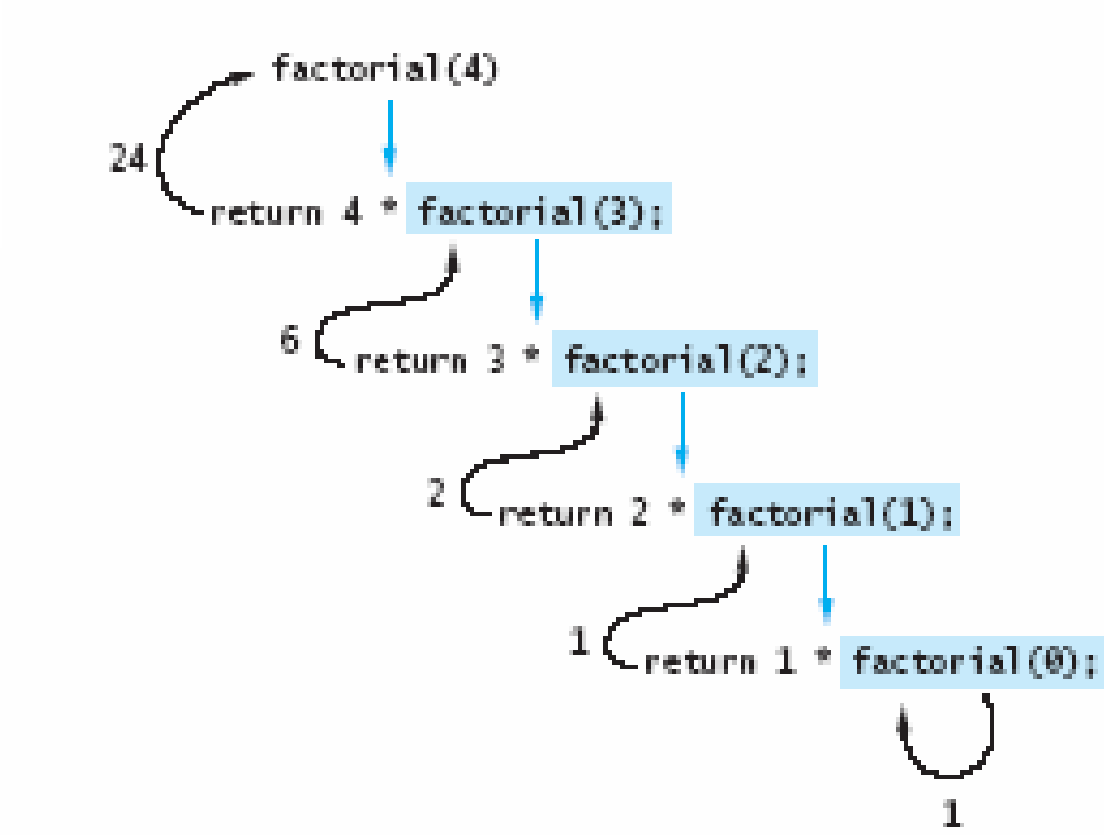


Recursive Definitions of Mathematical Formulas

- Mathematicians often use *recursive definitions*
- These lead very naturally to *recursive algorithms*
- Examples include:
  - Factorial
  - Powers
  - Greatest common divisor

Recursive Definitions: Factorial

- 0! = 1
- n! = n x (n-1)!



- If a recursive function never reaches its base case, a stack overflow error occurs

Recursive Definitions: Factorial Code

```
public static int factorial (int n) {
  if (n == 0) // or: throw exc. if < 0
    return 1;
  else
    return n * factorial(n-1);
}
```

Recursive Definitions: Power

- $x^0 = 1$
- $x^n = x \times x^{n-1}$

```
public static double power
    (double x, int n) {
  if (n <= 0)  // or: throw exc. if < 0
    return 1;
  else
    return x * power(x, n-1);
}
```

Recursive Definitions: Greatest Common Divisor

Definition of gcd(m, n), for integers m > n > 0:

- gcd(m, n) = n, if n divides m evenly
- gcd(m, n) = gcd(n, m % n), otherwise

```
public static int gcd (int m, int n) {
  if (m < n)
    return gcd(n, m);
  else if (m % n == 0) // could check n>0
    return n;
  else
    return gcd(n, m % n);
}
```

Recursion Versus Iteration

- Recursion and iteration are *similar*
- **Iteration:**
  - Loop repetition test determines whether to exit
- **Recursion:**
  - Condition tests for a base case
- Can always write iterative solution to a problem solved recursively, *but:*
- Recursive code often simpler than iterative
  - Thus easier to write, read, and debug

Tail Recursion → Iteration

When recursion involves <u>*single call*</u> that is <u>*at the end*</u> ...

It is called ***tail recursion*** and it easy to make iterative:

```
public static int iterFact (int n) {
  int result = 1;
  for (int k = 1; k <= n; k++) {
    result = result * k;
  }
  return result;
}
```

Efficiency of Recursion

- Recursive method often <u>*slower*</u> than iterative; ***why?***
  - Overhead for loop repetition smaller than
  - Overhead for call and return
- If easier to develop algorithm using recursion,
  - Then code it as a recursive method:
  - Software engineering benefit probably outweighs ...
  - Reduction in efficiency
- Don’t “optimize” prematurely!

Recursive Definitions: Fibonacci Series

Definition of fib_i, for integer $i > 0$:

- $fib_1 = 1$
- $fib_2 = 1$
- $fib_n = fib_{n-1} + fib_{n-2}$, for $n > 2$

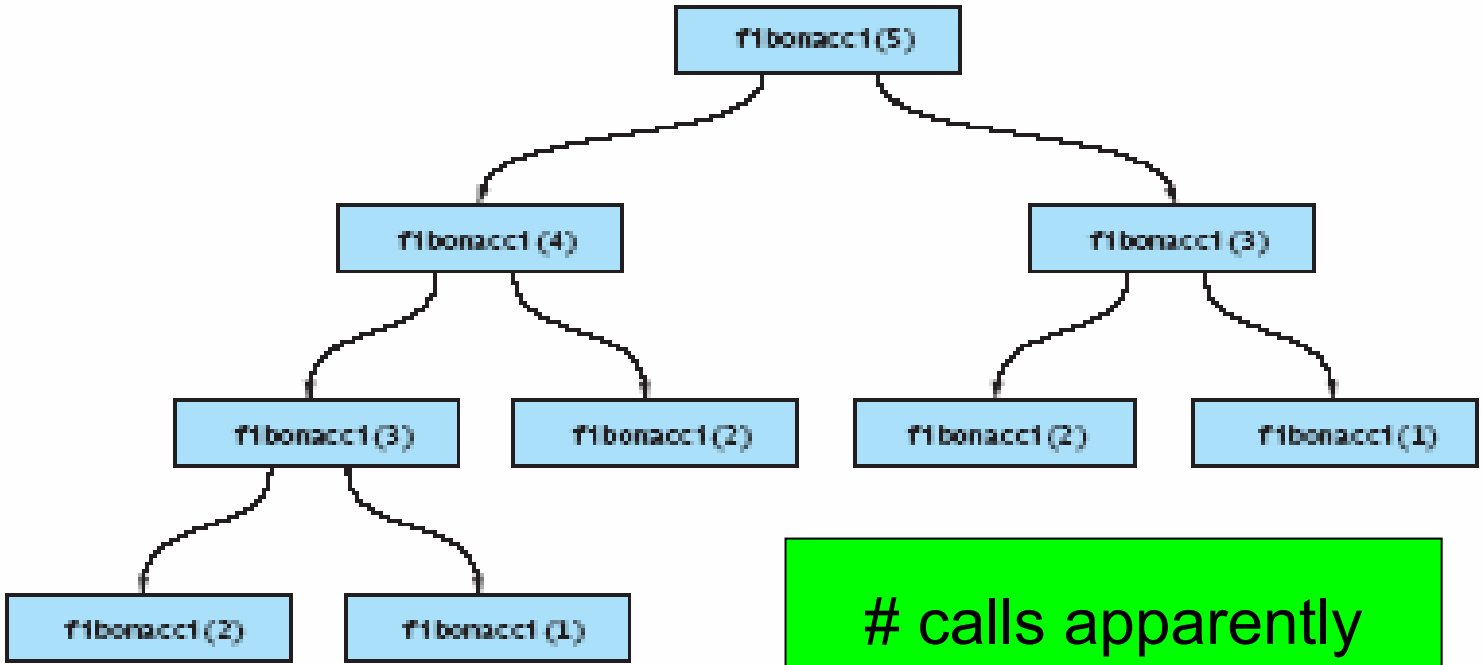
Fibonacci Series Code

```
public static int fib (int n) {
  if (n <= 2)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

This is straightforward, but an inefficient recursion ...

Efficiency of Recursion: Inefficient Fibonacci

FIGURE 7.6
Method Calls Resulting from `fibonacci(5)`



calls apparently $O(2^n)$ – big!

Efficient Fibonacci

- ***Strategy:*** keep track of:
  - *Current* Fibonacci number
  - *Previous* Fibonacci number
  - # left to compute

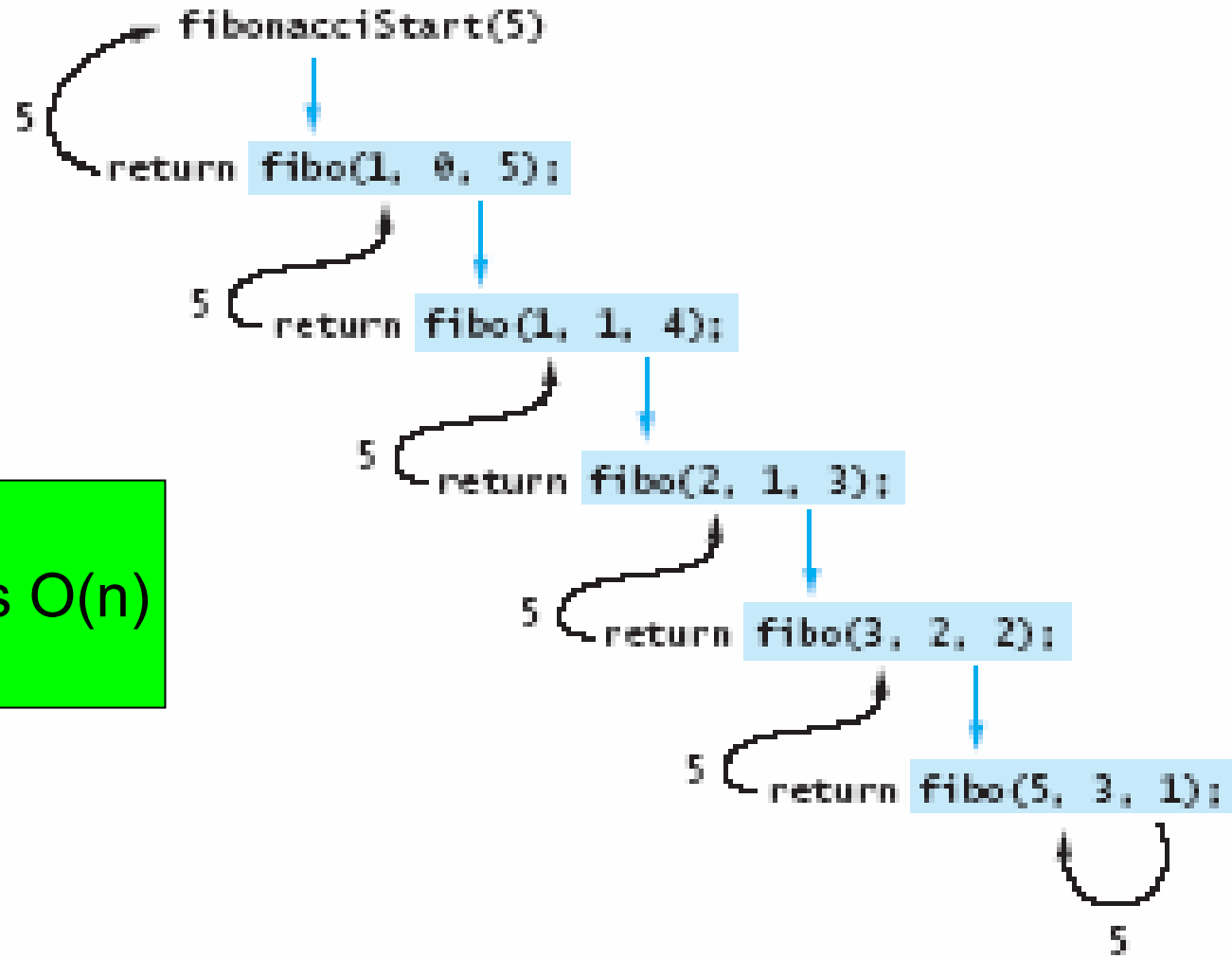
Efficient Fibonacci: Code

```
public static int fibStart (int n) {
  return fibo(1, 0, n);
}

private static int fibo (
    int curr, int prev, int n) {
  if (n <= 1)
    return curr;
  else
    return fibo(curr+prev, curr, n-1);
}
```

Efficient Fibonacci: A Trace

FIGURE 7.7
Trace of `fibonacciStart(5)`



Performance is O(n)

Recursive Array Search

- Can search an array using recursion
- Simplest way is linear search
  - Examine one element at a time
  - Start with the first element, end with the last
- One base case for recursive search: *empty array*
  - Result is -1 (negative, not an index → not found)
- Another base case: *current element matches target*
  - Result is index of current element
- Recursive case: *search rest, without current element*

Recursive Array Search: Linear Algorithm

```
if the array is empty
    return -1
else if first element matches target
    return index of first element
else
    return result of searching rest of the array,
        excluding the first element
```

Linear Array Search Code

```
public static int linSrch (
    Object[] items, Object targ) {
  return linSrch(items, targ, 0);
}

private static int linSrch (
    Object[] items, Object targ, int n) {
  if (n >= items.length) return -1;
  else if (targ.equals(items[n]))
    return n;
  else
    return linSrch(items, targ, n+1);
}
```

Linear Array Search Code: Alternate

```
public static int lSrch (
    Object[] items, Object o) {
  return lSrch(items, o, items.length-1);
}

private static int lSrch (
    Object[] items, Object o, int n) {
  if (n < 0) return -1;
  else if (o.equals(items[n]))
    return n;
  else
    return lSrch(items, targ, n-1);
}
```

Array Search: Getting Better Performance

- Item not found: O(n)
- Item found: n/2 on average, still O(n)
- How can we perhaps do better than this?
  - What if the array is sorted?
  - Can compare with middle item
  - Get two subproblems of size $\leq$ n/2
- What performance would this have?
  - Divide by 2 at each recursion → O(log n)
  - ***Much better!***

Binary Search Algorithm

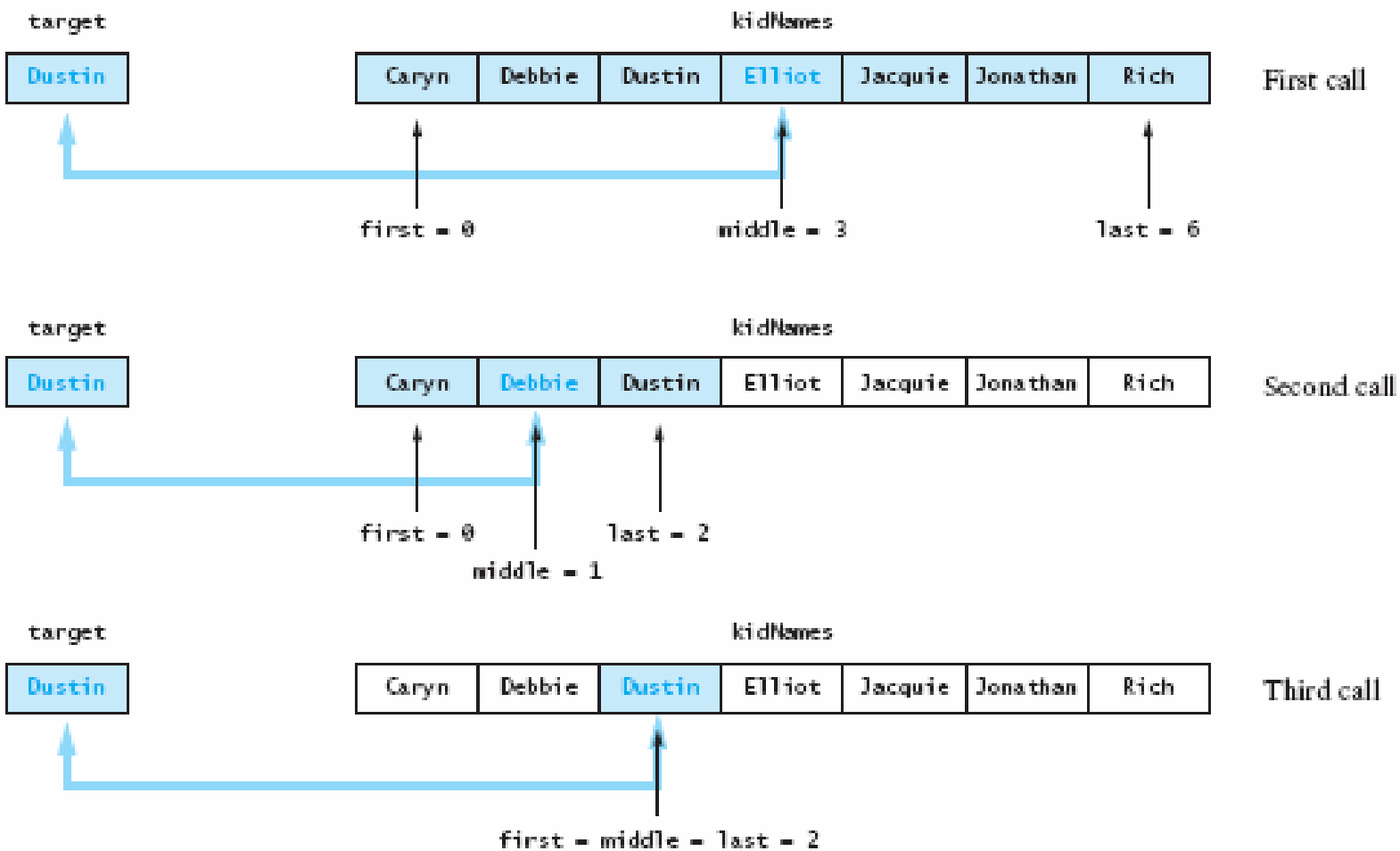
- Works only on *sorted* arrays!
- Base cases:
  - Empty (sub)array
  - Current element matches the target
- Check middle element with the target
- Consider only the array half where target can still lie

Binary Search Algorithm Steps

```
if array is empty
    return -1 as result
else if middle element matches
    return index of middle element as result
else if target < middle element
    return result of searching lower portion of array
else
    return result of searching upper portion of array
```

Binary Search Example

FIGURE 7.9
Binary Search of an Array



Binary Search Code

```
private static int bSrch (Object[] a,
    Comparable t, int lo, int hi) {
  if (lo > hi) // no elements
    return -1;
  int mid = (lo + hi) / 2;
  int comp = t.compareTo(a[mid]);
  if (comp == 0) // t equals mid element
    return mid;
  else if (comp < 0) // t < mid element
    return bSrch(a, t, lo, mid-1);
  else
    return bSrch(a, t, mid+1, hi);
}
```

Binary Search Code (2)

```
public static int bSrch (
    Object[] a, Comparable t) {
  return bSrch(a, t, 0, a.length-1);
}
```

Java API routine **`Arrays.binarySearch`** does this for:

- Sorted arrays of primitive types (**`int`**, etc.)
- Sorted arrays of objects
  - Objects must be **`Comparable`**

Recursive *Data Structures*

- Just as we have recursive *algorithms*
  - We can have recursive *data structures*
- Like algorithms, a recursive data structure has:
  - A *base case*, a simple data structure, or null
  - A recursive case: includes a *smaller* instance of the *same data structure*

Recursive Data Structures (2)

- Computer scientists often *define* data structures recursively
  - Trees (Chapter 8) are defined recursively
- Linked lists can also be defined recursively
- Recursive *methods* are very natural in processing recursive *data structures*

- The first language developed for artificial intelligence research was a recursive language called LISP

Recursive Definition of Linked List

A *linked list* is either:

- An *empty list* ← the base case, or
- A head node, consisting of: ← recursive case
  - A *data item* and
  - A reference to a *linked list* (rest of list)

Code for Recursive Linked List

```
public class RecLL<E> {
  private Node<E> head = null;
  private static class Node<E> {
    private E data;
    private Node<E> rest;
    private Node (E data, Node<E> rest) {
      this.data = data;
      this.rest = rest;
    }
  }
  ...
}
```

Code for Recursive Linked List (2)

```
private int size (Node<E> head) {
  if (head == null)
    return 0;
  else
    return 1 + size(head.next);
}

public int size () {
  return size(head);
}
```

Code for Recursive Linked List (3)

```
private String toString (Node<E> head) {
  if (head == null)
    return “”;
  else
    return toString(head.data) + “\n” +
           toString(head.next);
}

public String toString () {
  return toString(head);
}
```

Code for Recursive Linked List (4)

```
private Node<E> find (
    Node<E> head, E data) {
  if (head == null)
    return null;
  else if (data.equals(head.data))
    return head;
  else
    return find(head.next, data);
}

public boolean contains (E data) {
  return find(head, data) != null;
}
```

Code for Recursive Linked List (5)

```
private int indexOf (
      Node<E> head, E data) {
  if (head == null)
    return -1;
  else if (data.equals(head.data))
    return 0;
  else
    return 1 + indexOf(head.next, data);
}

public int indexOf (E data) {
  return indexOf(head, data);
}
```

Code for Recursive Linked List (6)

```
private void replace (Node<E> head,
    E oldE, E newE) {
  if (head == null)
    return;
  else {// replace all old: always recurse
    if (oldE.equals(head.data))
      head.data = newE;
    replace(head.next, oldE, newE);
  }
}
public void replace (E oldE, E newE) {
  replace(head, oldE, newE);
}
```

Code for Recursive Linked List (7)

```
private void add (Node<E> head, E data) {
  // Note different base case!!
  if (head.next == null)
    head.next = new Node<E>(data);
  else // replace all old: always recurse
    add(head.next, data);
}
public void add (E data) {
  if (head == null)
    head = new Node<E>(data);
  else
    add(head, data);
}
```

Code for Recursive Linked List (8)

```
private boolean remove (
    Node<E> pred, Node<E> curr, E data) {
  if (curr == null) {  // a base case
    return false;
  } else if (data.equals(curr.data)) {
    pred.next = curr.next;
    return true; // 2d base case
  } else {
    return remove(curr, curr.next, data);
  }
}
```

Code for Recursive Linked List (9)

```
public boolean remove (E data) {
  if (head == null) {
    return false;
  } else if (data.equals(head.data)) {
    head = head.next;
    return true;
  } else {
    return remove(head, head.next, data);
  }
}
```

Alternate Recursive Linked List

```
private Node<E> add (
    Node<E> head, E data) {
  if (head == null)
    return new Node<E>(data);
  else
    return new Node<E>(
      data, add(head.next, data));
  // more elegant; more allocation
}
public void add (E data) {
  head = add(head, data);
}
```

Alternate Recursive Linked List (2)

```
private Node<E> add (
    Node<E> head, E data) {
  if (head == null)
    return new Node<E>(data);
  else {
    head.next = add(head.next, data);
    return head;
  }
}
public void add (E data) {
  head = add(head, data);
}
```

Alternate Recursive Linked List (3)

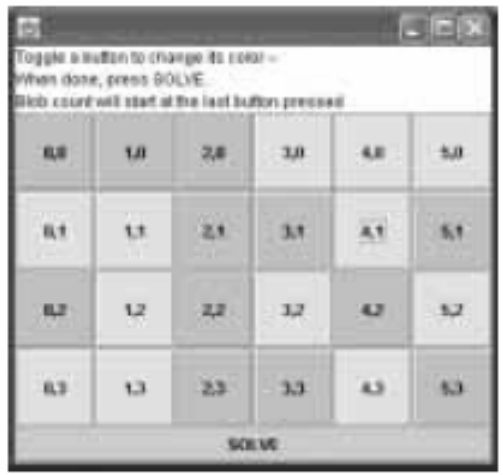
```
private Node<E> remove (
      Node<E> head, E data) {
  if (head == null) return null;
  else if (data.equals(head.data))
    return remove(head.next, data);
  else {
    head.next = remove(head.next, data);
    return head;
  }
}
public void remove (E data) {
  head = remove(head, data);
}
```

Problem Solving with Recursion

- Towers of Hanoi
- Counting grid squares in a blob
- Backtracking, as in maze search

FIGURE 7.11
Children's Version of Towers of Hanoi

FIGURE 7.16
A Sample Grid for Counting Cells in a Blob



Towers of Hanoi: Description

Goal: Move entire tower to another peg

Rules:

1. You can move only the top disk from a peg.
2. You can only put a smaller on a larger disk (or on an empty peg)

FIGURE 7.11
Children's Version of Towers of Hanoi

Towers of Hanoi: Solution Strategy

FIGURE 7.11
Children's Version of Towers of Hanoi

FIGURE 7.12
Towers of Hanoi After the First Two Steps in Solution of the Three-Disk Problem

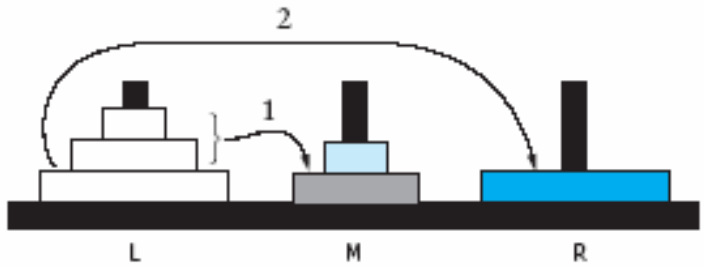
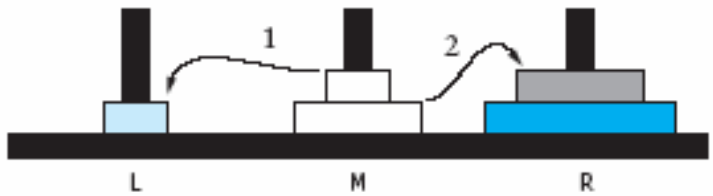


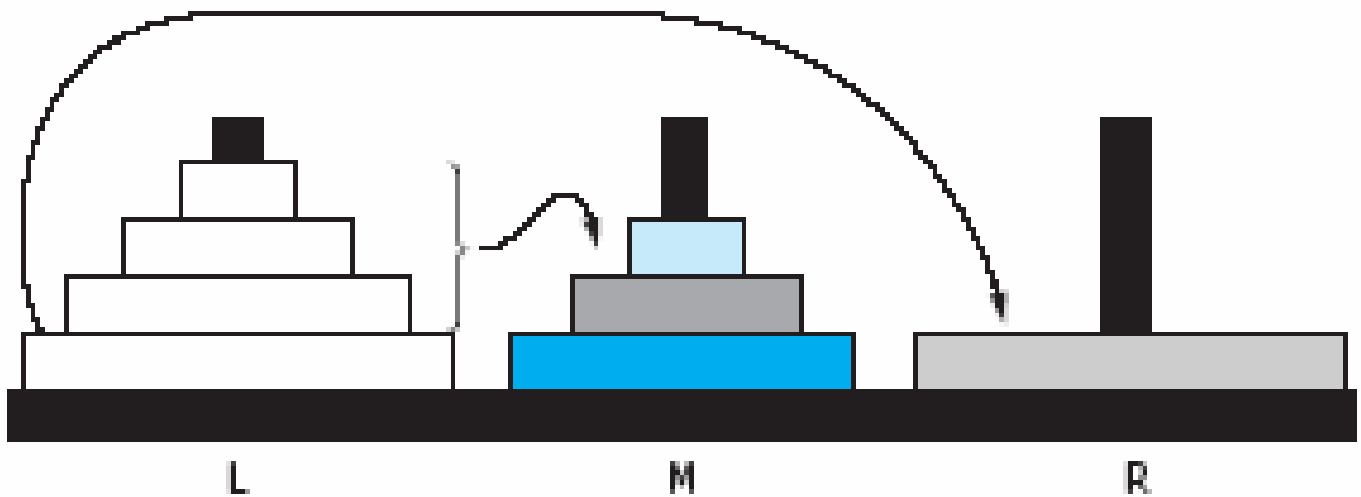
FIGURE 7.13
Towers of Hanoi After First Two Steps in Solution of Two-Disk Problem



Towers of Hanoi: Solution Strategy (2)

FIGURE 7.14

Towers of Hanoi After the First Two Steps in Solution of the Four-Disk Problem



Towers of Hanoi: Program Specification

TABLE 7.1

Inputs and Outputs for Towers of Hanoi Problem

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

Towers of Hanoi: Program Specification (2)

TABLE 7.2
Class `TowersOfHanoi`

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.

Towers of Hanoi: Recursion Structure

```
move(n, src, dst, tmp) =
  if n == 1:  move disk 1 from src to dst
  otherwise:
      move(n-1, src, tmp, dst)
      move disk n from src to dst
      move(n-1, tmp, dst, src)
```

Towers of Hanoi: Code

```
public class TowersOfHanoi {
  public static String showMoves(int n,
      char src, char dst, char tmp) {
    if (n == 1)
      return “Move disk 1 from “ + src +
             “ to “ + dst + “\n”;
    else return
      showMoves(n-1, src, tmp, dst) +
      “Move disk “ + n + “ from “ + src +
        “ to “ + dst + “\n” +
      showMoves(n-1, tmp, dst, src);
  }
}
```

Towers of Hanoi: Performance Analysis

How big will the string be for a tower of size n?

We'll just count lines; call this L(n).

- For n = 1, one line: L(1) = 1
- For n > 1, one line plus twice L for next smaller size:

  $L(n+1) = 2 \times L(n) + 1$

Solving this gives $L(n) = 2^n - 1 = O(2^n)$

So, don't try this for very large n – you will do a lot of string concatenation and garbage collection, and then run out of heap space and terminate.

Counting Cells in a Blob

- *Desire:* Process an image presented as a two-dimensional array of color values
- Information in the image may come from
  - X-Ray
  - MRI
  - Satellite imagery
  - Etc.
- *Goal:* Determine size of any area considered abnormal because of its color values

Counting Cells in a Blob (2)

- A *blob* is a collection of *contiguous* cells that are *abnormal*
- By *contiguous* we mean cells that are adjacent, horizontally, vertically, or diagonally

Counting Cells in a Blob: Example

FIGURE 7.16

A Sample Grid for Counting Cells in a Blob

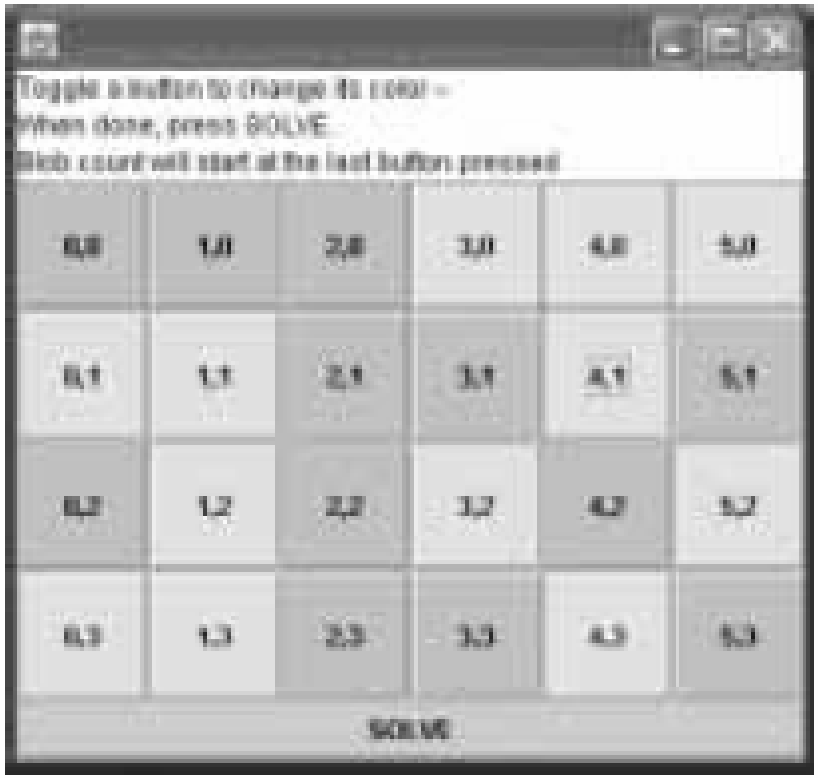
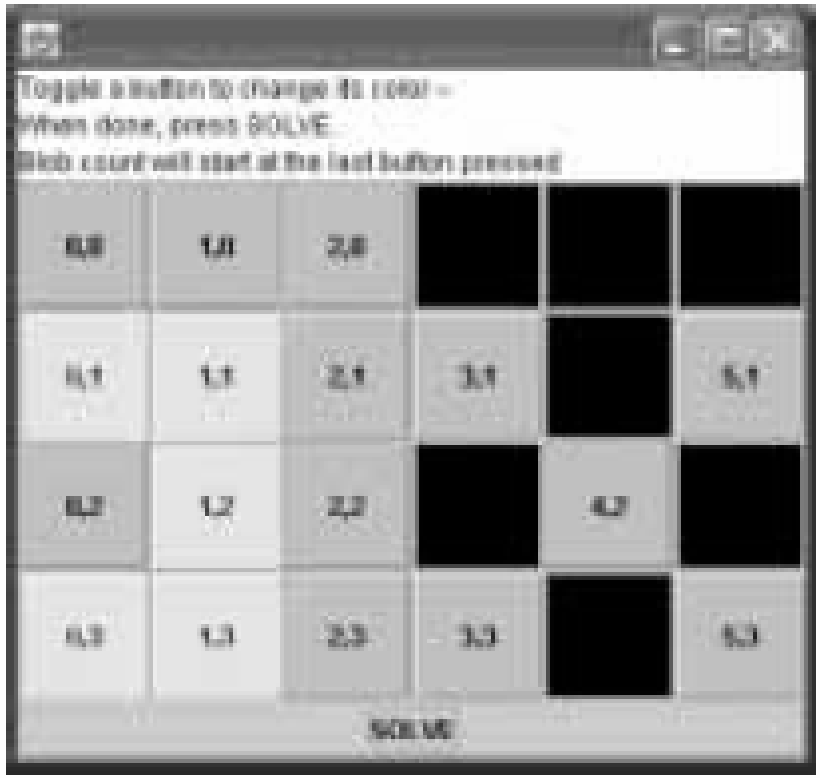


FIGURE 7.17

Blob Cells (in Black) After Execution of `countCells`



Counting Cells in a Blob: Recursive Algorithm

```
Algorithm countCells(x, y):
if (x, y) outside grid
        return 0
else if color at (x, y) normal
        return 0
else
        Set color at (x, y) to “Temporary” (normal)
        return 1 + sum of countCells on neighbors
```

Counting Cells: Program Specification

TABLE 7.3
Class `TwoDimGrid`

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.

TABLE 7.4
Class `Blob`

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

Count Cells Code

```
public class Blob implements GridColors {
  private TwoDimGrid grid;

  public Blob(TwoDimGrid grid) {
    this.grid = grid;
  }

  public int countCells(int x, int y) {
    ...
  }
}
```

Count Cells Code (2)

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

Backtracking

- *Backtracking:* *systematic* trial and error search for solution to a problem
  - *Example:* Finding a path through a maze
- In walking through a maze, probably walk a path as far as you can go
  - Eventually, reach destination or dead end
  - If dead end, must retrace your steps
  - Loops: stop when reach place you've been before
- Backtracking systematically tries alternative paths and eliminates them if they don't work

Backtracking (2)

- If you never try exact same path more than once, and
- You try all possibilities,
- You will eventually find a solution path if one exists

- Problems solved by backtracking: a set of *choices*
- Recursion implements backtracking straightforwardly
  - Activation frame remembers choice made at that decision point
- A chess playing program likely involves backtracking

Maze Solving Algorithm: findPath(x, y)

```
if (x,y) outside grid, return false
if (x,y) barrier or visited, return false
if (x,y) is maze exit, color PATH and return true
else:
    set (x,y) color to PATH ("optimistically")
    for each neighbor of (x,y)
        if findPath(neighbor), return true
    set (x,y) color to TEMPORARY ("visited")
    return false
```

Maze Solving Code

```
public class Maze implements GridColors {
  private TwoDimGrid maze;
  public Maze (TwoDimGrid maze) {
    this.maze = maze;
  }
  public boolean findPath() {
    return findPath(0, 0);
  }
  public boolean findPath (int x, int y) {
    ...
  }
}
```

Maze Solving Code (2)

```
public boolean findPath (int x, int y) {
  if (x < 0 || x >= maze.getNCols() ||
      y < 0 || y >= maze.getNRows())
    return false;
  Color xyColor = maze.getColor(x,y);
  if (!xyColor.equals(BACKGROUND))
    return false;
  maze.recolor(x, y, PATH);
  if (x == maze.getNCols() – 1 &&
      y == maze.getNRows() – 1)
    return true;
  ...
```

Maze Solving Code (3)

```
  // square ok, but not end;
  // it's colored PATH (tentatively)
  if (findPath(x-1, y  ) ||
      findPath(x+1, y  ) ||
      findPath(x  , y-1) ||
      findPath(x  , y+1))
    return true;

  // a dead end: mark visited and return
  maze.recolor(x, y, TEMPORARY);
  return false;
}
```