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

Chapter Objectives (continued)

- 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*** is a problem-solving approach in which a problem is solved using repeatedly applying the same solution to smaller instances.
  - Each instance to the problem has size.
  - An instance of size *n* can be solved by putting together solutions of instances of size at most *n*-1.
  - An instance of size 1 or 0 can be solved very easily.

An Example: Computing the Length of a List Object

- Two data fields: data and next.
- If the element is null, return 0.
- If the element is not null, but the next is null, return 1.
- Otherwise, return 1 + the length of the list starting with the next.

```
public int count() {
        if (this == null) return 0;              // Base case 0
        else if (next == null) return 1;         // Base case 1
        else return 1 + next.count();            // Recursive call
}
```

Proof of Correctness is Similar to Proof-By-Induction

- Proof by induction
  - Prove the statement is true for the base case (size 0,1, or whatever).
  - Show that if the statement is assumed true for n, then it must be true for n+1.

Proof of Correctness

- Recursive proof is similar to induction. Verify that:
  - The base case is recognized and solved correctly
  - Each recursive case makes progress towards the base case
  - If all smaller problems are solved correctly, then the original problem is also solved correctly

System Processing of a Recursive Call

- Push onto a stack the information of the current execution.
- Execute the recursive call.
- Retrieve the information from the stack by pop.
- Too many recursive calls without pop will result in stack overflow.

Recursively Defined Mathematical Functions

Recursive Definitions of Mathematical Formulas

- ***Factorial***: n! where n >= 0.
  - 0! = 1.
  - If $n > 0$, $n! = n \times (n-1)!$.
- ***Powers***: x^n, x to the power of n, where $x > 0$, $n >= 0$.
  - If n=0, $x^n = 1$.
  - If n>0, $x^n = x \times x^{n-1}$.
- ***Greatest Common Divisor***: gcd(a,b), a, b >= 0
  - gcd(a,b) = gcd(b,a).
  - If b=0, gcd(a,b) = a.
  - If $a >= b$, gcd(a,b) = gcd(a-b,b).

Factorial, Powers, and gcd

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

public static int powers(int x, int n) {
        if ((x <= 0) || (n < 0)) return 0;
        else if (n == 0) return 1;
        else return n * powers(x, n-1);
}

public static int gcd(int a, int b) {
        if ((a<0) || (b<0)) return 0;
        else if (a == 0) return b;
        else if (b == 0) return a;
        else if (a < b) return gcd(b,a);
        else return gcd(b, a % b);
}
```

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
- An iterative solution exists to a problem that is solvable by recursion
- Recursive code may be simpler than an iterative algorithm and thus easier to write, read, and debug

Iterative Solutions

```
public static int factorial(int n) {
        if (n<0) return 0;
        fac = 1;
        for (int i=0; i<n; i++) { fac *= i; }
        return fac;
}
public static int powers(int x, int n) {
        if ((x <= 0) || (n < 0)) return 0;
        else if (n == 0) return 1;
        pow = 1;
        for (int i=0; i<n; i++) { pow *= x; }
        return pow;
}
public static int gcd(int a, int b) {
        if ((a<0) || (b<0)) return 0;
        if (a < b) { c = a; a = b; b = c;            // swap a and b.
        while (b>0) {
                r = a % b; a = b; b = r;             // Reduce a&b to b&(a mod b).
        }
        return a;
}
```

Efficiency of Recursion

- Recursive methods are often than iterative methods because the stack overhead is larger than the loop overhead.
- Recursive methods are easier to write and conceptualize.

Fibonacci Number

- $F(1) = 1$.
- $F(2) = 1$.
- For $n>=3$, $F(n) = F(n-1) + F(n-2)$

An O(2^n) Recursive Method

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

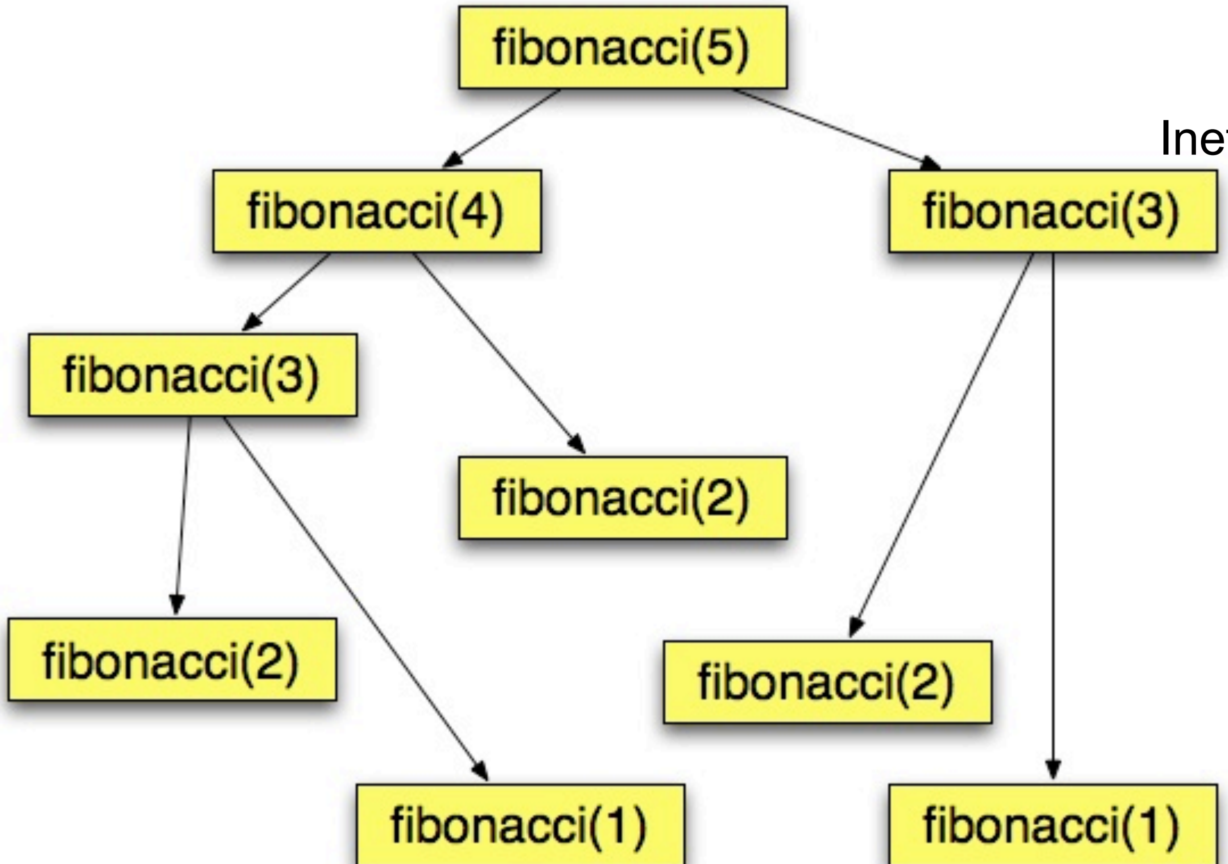
An O(n) Recursive Method

- fibo(a,b,c) is invoked to compute F(c+m) when F(m+1) =a and F(m)=b

```
public static int fibo(int fiboCurrent, int fiboPrevious, int c) {
        if (c == 1) return fiboCurrent;
        else return fibo(fiboPrevious+fiboCurrent, fiboCurrent, c-1);
}
```

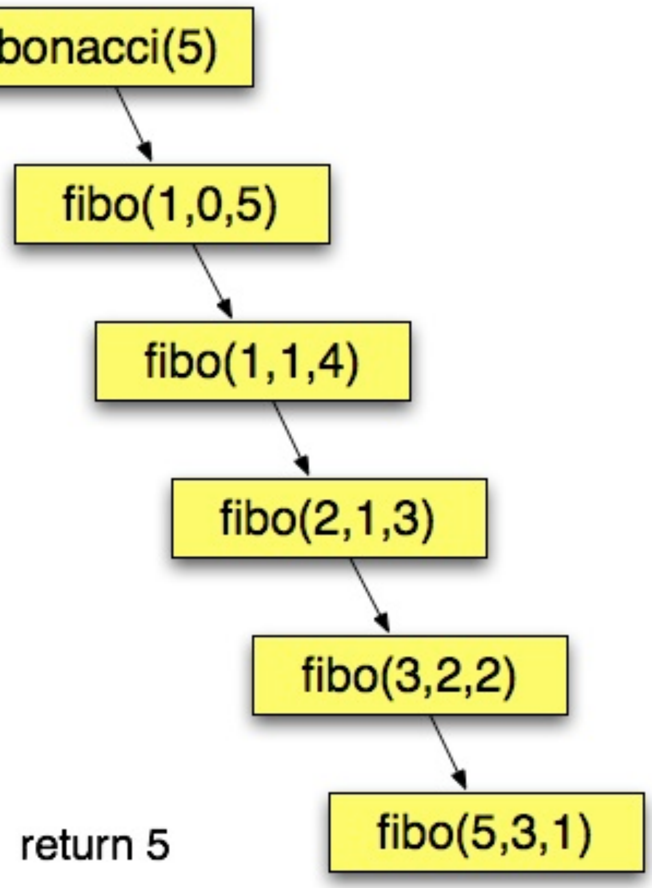
Invoke fibo(1,0,n)

Efficiency of Recursion: Exponential Fibonacci



Inefficient

Efficiency of Recursion: O(n) Fibonacci



Recursive Array Search

Linear Search

Versus

Binary Search

Linear Search in an Array A of size S

- To search A for an element E, invoke the search in the range [0, S-1].
  - To search in the range [I, S-1]
    - If I == S, then stop – the element is not in the array.
    - If the I-th element in the array is the one, stop there.
    - Otherwise, recursively search in the range[I+1,S-1].

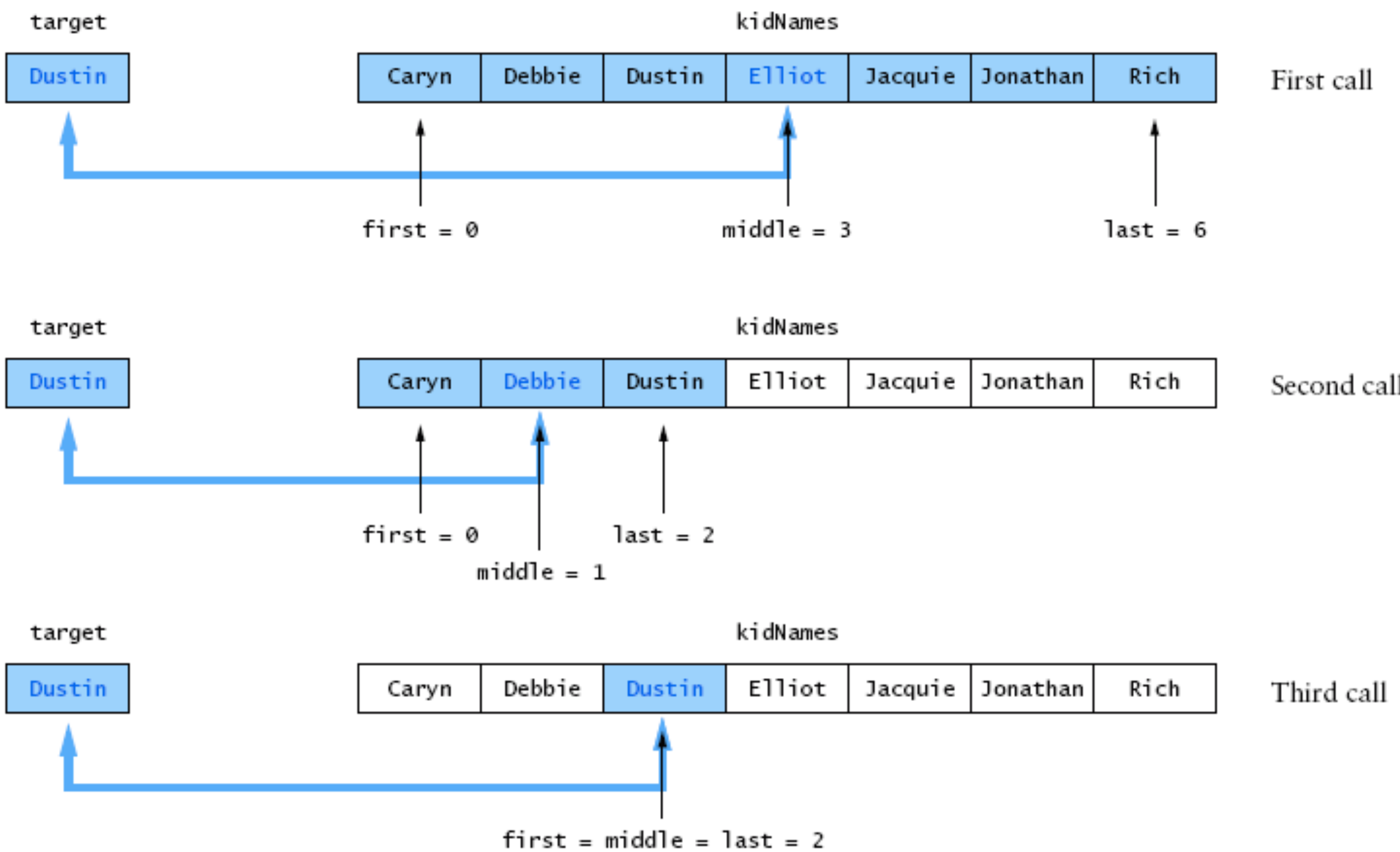
- Search requires O(S) time.

Binary Search in a Sorted Array

- The sorted elements in an array A. The first element is the smallest and the last element is the largest.
- Maintain the range of indices [I,J] for search.
- Loop:
  - Indicate “Not found” if I>J.
  - Let K be the middle index; the integer part of (I+J)/2.
  - If (A[K] == target), indicate “Found”.
  - Else if (A[K] > target), set J to K-1.
  - Else set I to K+1.
- At each iteration, the size of range becomes at most a half, so the running time is $O(\log_2 n)$.

Design of a Binary Search Algorithm (continued)

FIGURE 7.9
Binary Search for "Dustin"



Algorithm for Binary Search

```
/** search items for target in the range [first, last]
 * returns the index of the target if found; -1 o.w.
 */
public int binarySearch(Object[] items, Comparable target, int first, int last) {
        if (first>last) return -1;
        int mid = (first + last)/2;
        if (target.compareTo(items[mid])==0) {
                return mid;
        }
        else if (target.compareTo(items[mid]) < 0) { }
                return binarySearch(items, target, first, mid-1);
        }
        else {
                return binarySearch(items, target, mid+1, last);
        }
}
```

- The target has to be of a data type that implements compareTo:
    - CompareTo is a method that gives as an integer the ordering between two elements (usually -1, 0, 1).

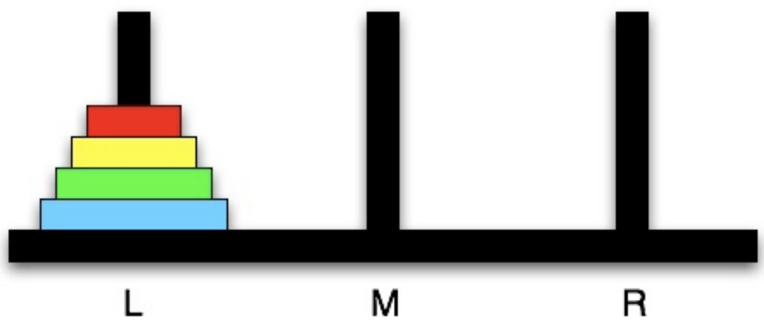
Method Arrays.binarySearch

- Java API class Arrays contains a binarySearch method
  - Can be 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

The Tower Of Hanoi

Towers of Hanoi

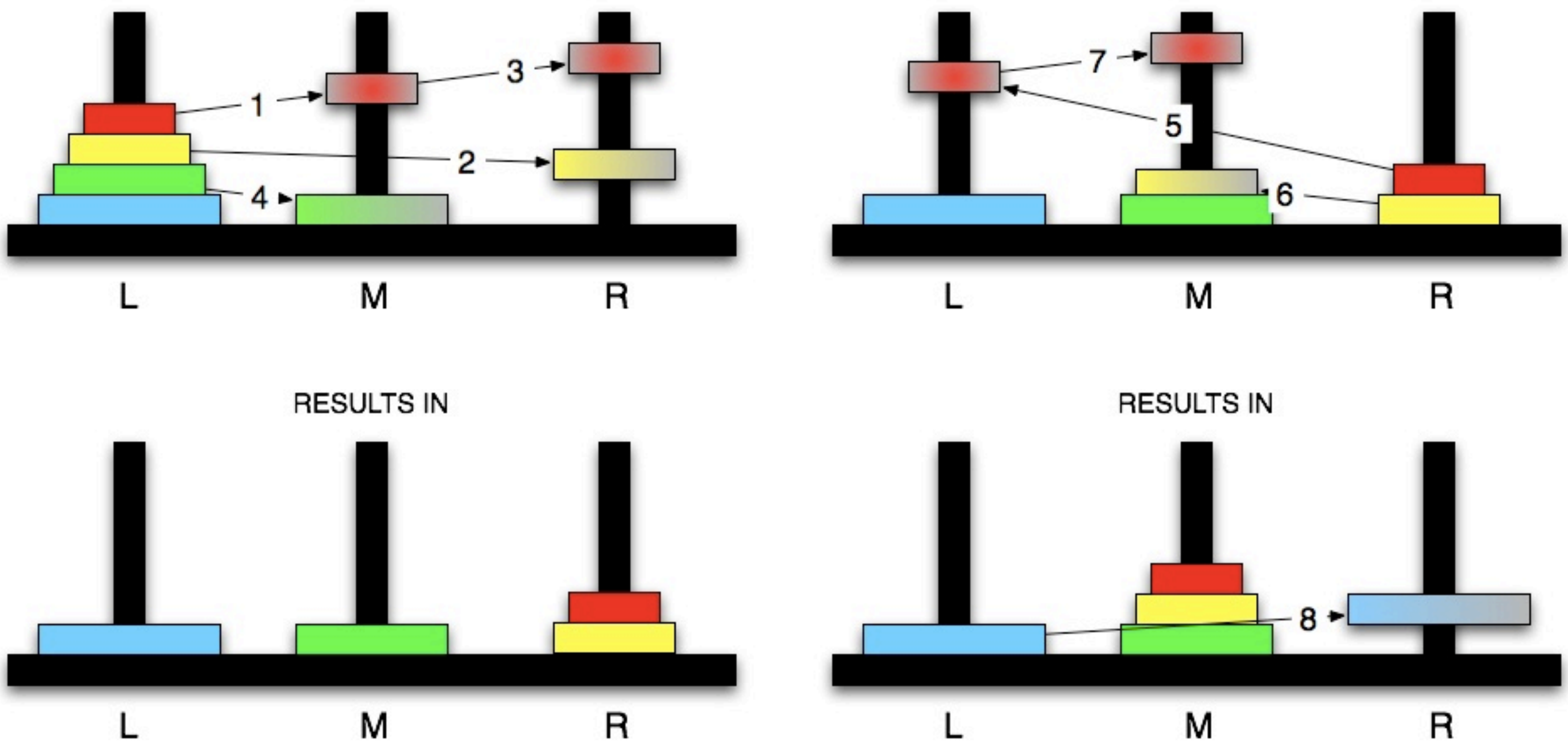
- There are 64 discs of all distinct diameters slid onto a peg in the increasing order of diameters with the smallest one on the very top. There are two other pegs
- Move all the discs to one of the two other pegs with the following rules:
  - A disc can be moved only one at a time.
  - A larger disc must not be placed on a smaller disc.
- Goal: Compute the shortest moves to accomplish this talk.



Formulation of the Towers-of-Hanoi Problem

- Consider the sub-problem of moving the top N discs from peg X to peg Y, where $1 <= N <= 64$, $X \neq Y$, and X and Y are members of { L, M, R }.
- The initial invocation: N = 64, X=L, Y=R/M.

Algorithm for Towers of Hanoi: N=4



Recursive Algorithm for Towers of Hanoi

- Input N, X, Y
- N=1:
  - Move the top disc of peg X to peg Y.
- N>1:
  - Let Z be the peg other than X or Y.
  - Move the top N-1 discs from X to Z.
  - Move the top 1 disc from X to Y.
  - Move the top N-1 discs from Z to Y.

Towers of Hanoi Class

```
public class TowersOfHanoi {
	/** Recursive method for Towers of Hanoi
		pre: the three chars are all distinct
		@param n is the number of disks
		@param startPeg is the peg where the disks currently are
		@param destPeg is the peg which the disks should move to
		@param tempPeg is the remaining peg
	public static String showMoves(int n, char startPeg, char destPeg,
						char tempPeg) {
		if (n==1) {
			return
			“Move disc 1 from ” + startPeg + “ to ” + destPeg + “\n”;
		} else {
			return
			TowersOfHanoi(n-1, startPeg, tempPeg, destPeg) +
			“Move disc ” + n + “ from ” + startPeg + “ to ” + destPeg + “\n”;
			TowersOfHanoi(n-1, tempPeg, destPeg, startPeg);
		}
	}
}
```

Backtracking

Backtracking

- Backtracking is an approach to implementing systematic trial and error in a search for a solution.
  - It explores alternative search paths and eliminates them if they don’t work.
  - It remembers the search history to avoid trying the same path again.
- Recursion is a natural way to implement backtracking
  - The trace of successive recursive calls represents the search history with an instrumentation of exhaustive search at each level.

Maze Threading

- Input: a two-dimensional of cells, M by N
  - BACKGROUND … a cell can be walked in
  - BLOCKED … a cell that can never be walked in
- Output:
  - A path from (0,0) to (M-1, N-1) that visits only BACKGROUND cells, if one exists
- Additional Types:
  - PATH … a cell that is determined to be on the path to be built
  - TEMPORARY … a non-blocked cell that is found not be on any path to the goal

Algorithm for Maze Threading

- Use recursive search from cell (I,J)

if (I<0 || J<0 || I>=M || J>=N) { **return** false; }
else if (type of cell (I,J) != BACKGROUND) { **return** false; }
else {
  set type of cell (I,J) to PATH;
  if (I,J) is goal { **return** true; }
  else if Search from (I-1,J) returns true { **return** true; }
  else if Search from (I+1,J) returns true { **return** true; }
  else if Search from (I,J-1) returns true { **return** true; }
  else if Search from (I,J+1) returns true { **return** true; }
  else { set type of cell (I,J) to TEMPORARY; **return** false; } }