Class 06 - Recursion

CSIS 3475

Data Structures and Algorithms

What Is Recursion?

- Consider hiring a contractor to build
  - He hires a subcontractor for a portion of the job
  - That subcontractor hires a sub-subcontractor to do a smaller portion of job
- The last sub-sub- … subcontractor finishes
  - Each one finishes and reports “done” up the line

Example: The Countdown from 10

Counting down recursively

- How does it stop? When count is at 1.

```
public class Countdown {

    public static void main(String[] args) {
        countDown(10);
    }

    public static void countDown(int integer) {
        System.out.println(integer);
        if (integer > 1)
            countDown(integer - 1);
    }
}
```

Definition

- Recursion is a problem-solving process
  - Breaks a problem into identical but smaller problems.
- A method that calls itself is a recursive method.
  - The invocation is a recursive call or recursive invocation.

Design Guidelines

- Method must be given an input value
- Method definition must contain logic that involves this input, leads to different cases
- One or more cases should provide solution that does not require recursion
  - Else infinite recursion
  - Stopping case
- One or more cases must include a recursive invocation

Programming Tip

- Iterative method contains a loop
- Recursive method calls itself
- Some recursive methods contain a loop and call themselves
    - If the recursive method with loop uses while, make sure you did not mean to use an if statement

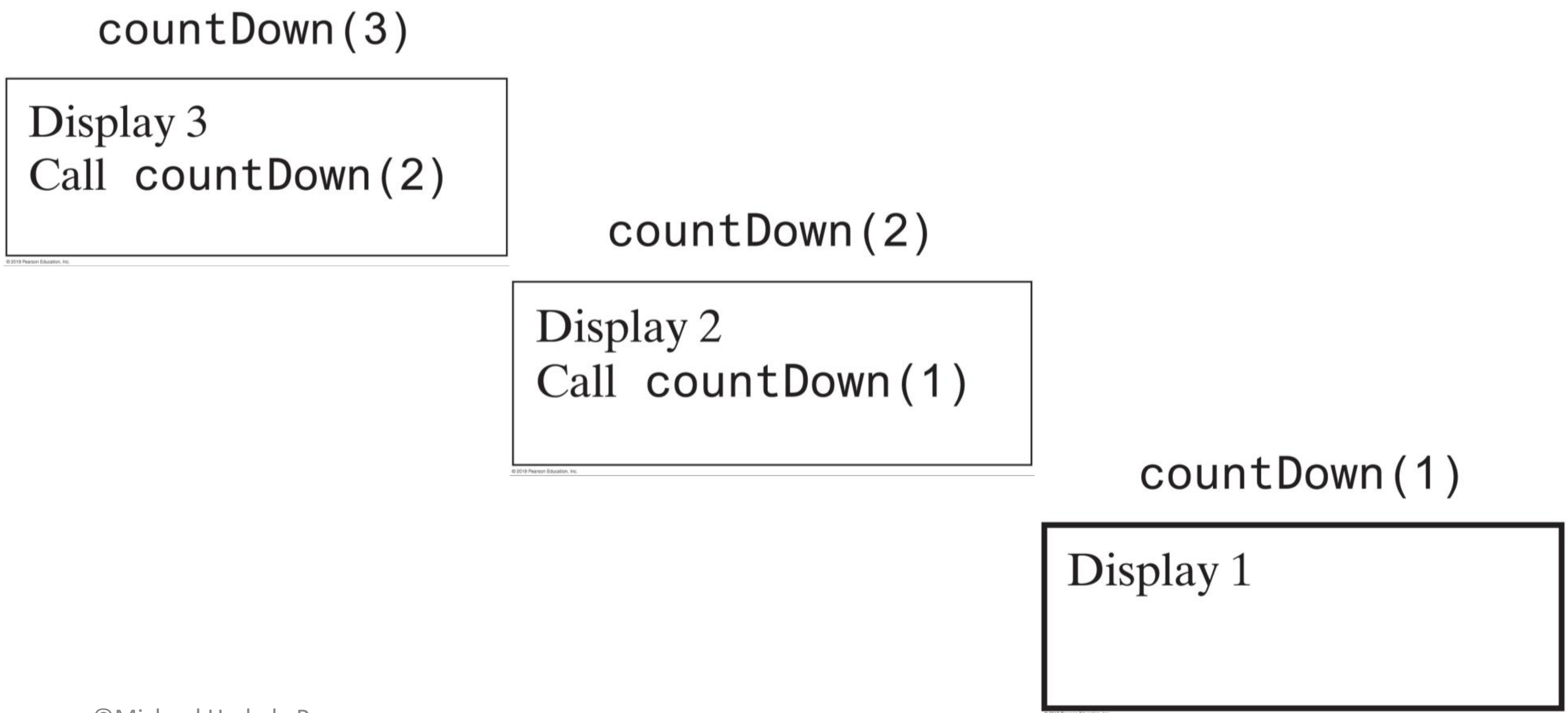
Countdown

- Countdown contains a method which calls itself (recursion)
- Contains a stopping condition

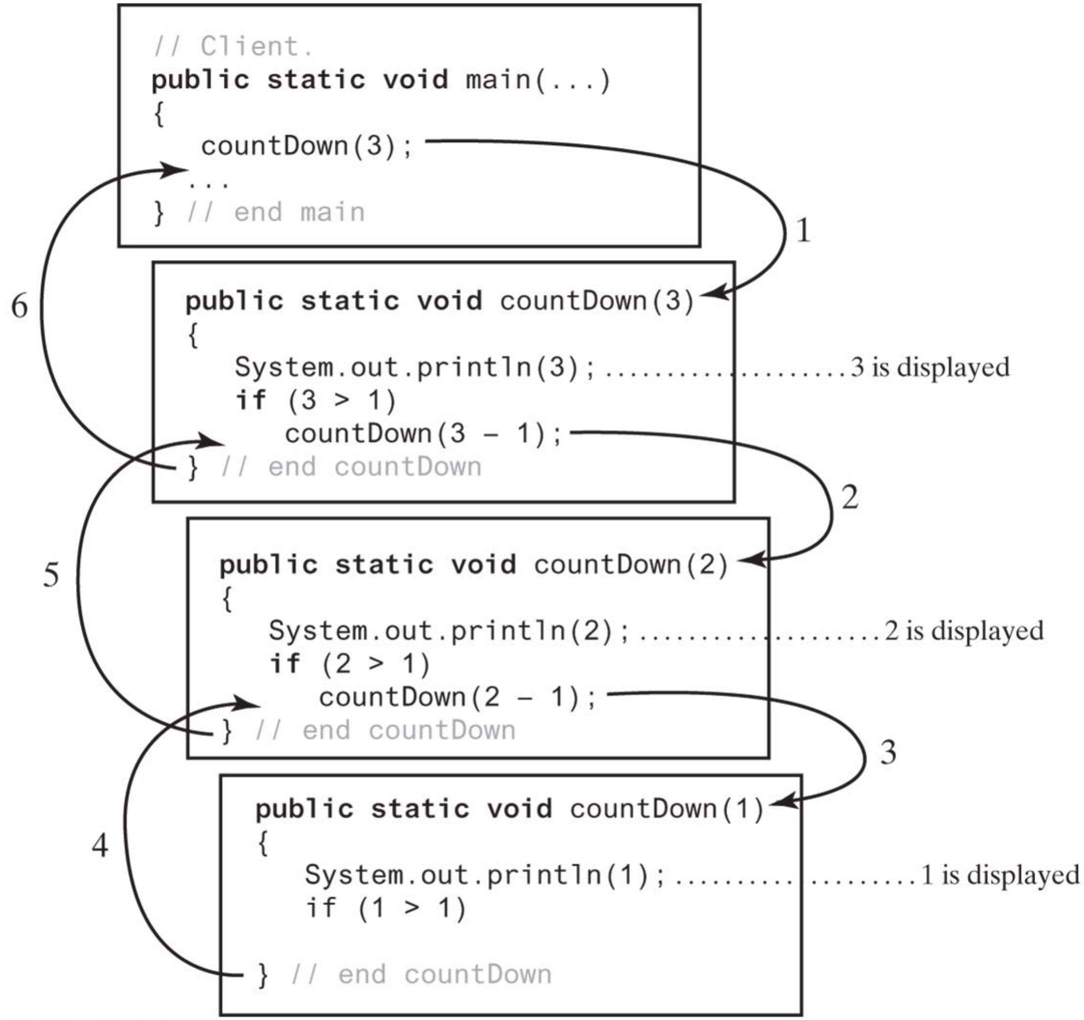
```
public class Countdown {

	public static void main(String[] args) {
		countDown(10);
	}

	public static void countDown(int integer) {
		System.out.println(integer);
		if (integer > 1)
			countDown(integer - 1);
	}
}
```

Tracing a Recursive Method

- The effect of the method call `countDown(3)`



Tracing the execution of `countDown(3)`

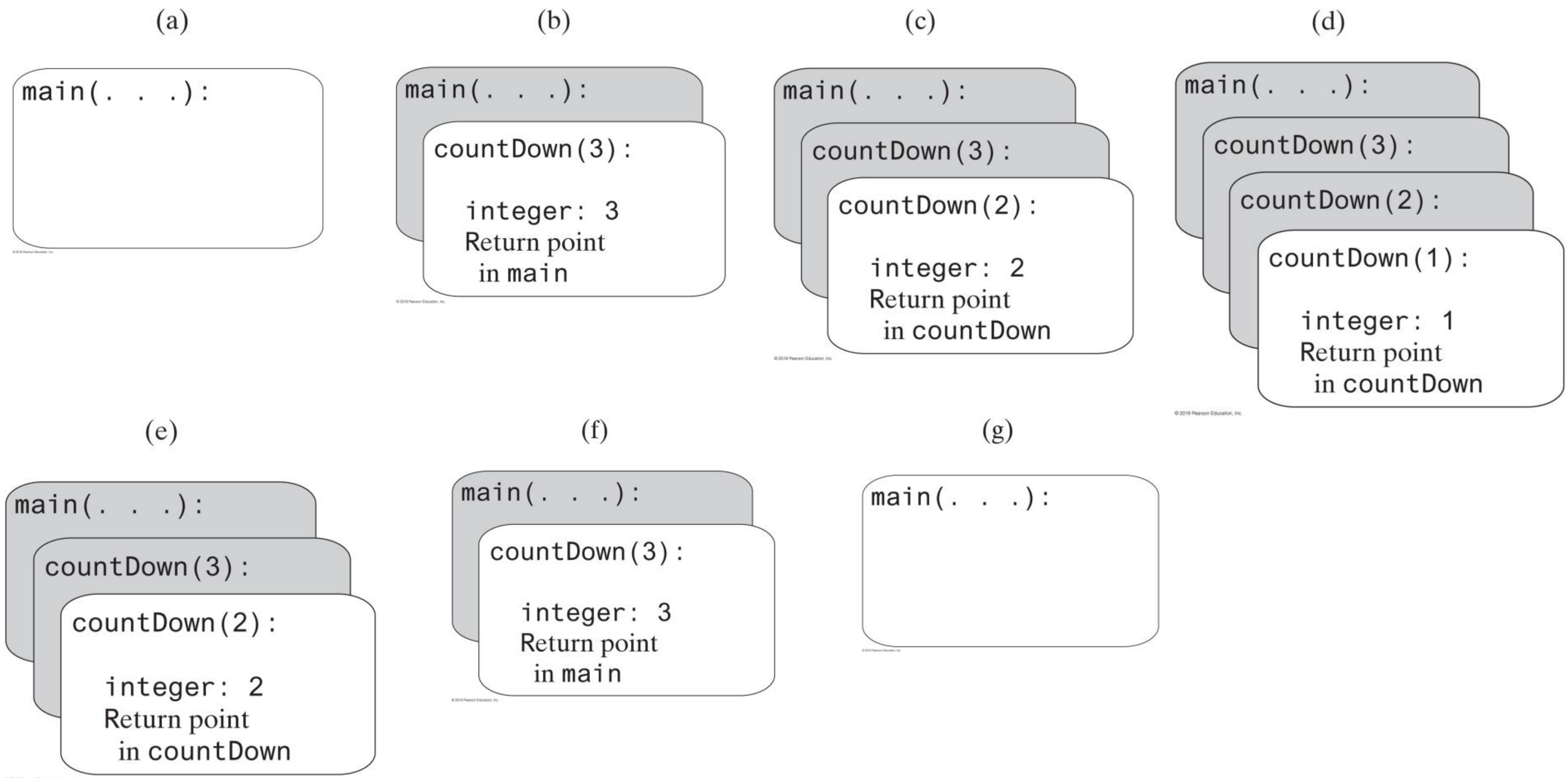


Stack of Activation Records

- Each call to a method generates an activation record
- Recursive method uses more memory than an iterative method
  - Each recursive call generates an activation record
- If recursive call generates too many activation records, could cause stack overflow

Stack of Activation Records

- The stack of activation records during the execution of the call `countDown(3)`



Sum of a series: O(1) non-recursive

- Let
  - S=1+2+...+(n−1)+n.
- Write it backwards
  - S=n+(n−1)+...+2+1.
- Add the two equations, term by term. Each term is n+1, so
  - 2S=(n+1)+(n+1)+...+(n+1)=n(n+1).
- Divide by 2:
  - S=n(n+1)/2
- Attributed to Gauss

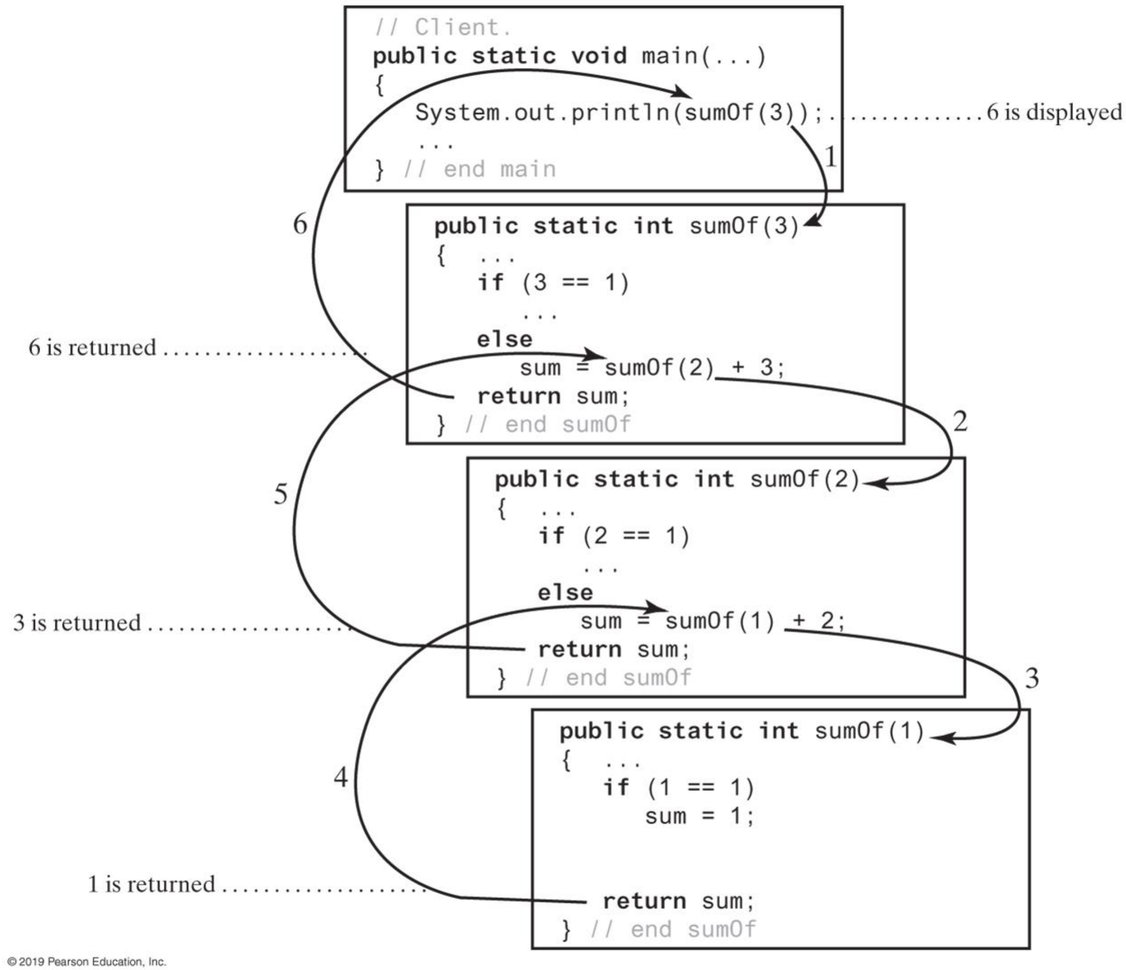
```
System.out.println("Sum of Series by algebra of " + input + " = " + (input * (input + 1) / 2));
```

Iterative vs Recursive sum of series

```java
/**
 * Compute sum of integer series using a loop
 * @param n last term of integer series
 * @return sum of series
 */
static int sumOfSeriesIterative(int n) {
	int sum = 0;

	for(int i = 1; i <= n; i++)
		sum += i;
	return sum;
}

/**
 * Compute sum of integer series using basic recursion
 *
 * @param n last term of integer series
 * @return sum of series
 */
static int sumOfSeriesRecursion(int n) {
	System.out.println("SumOfSeries(" + n + ")");

	// base case sum of 1 is 1
	if (n == 1)
		return 1;
	else {
		// otherwise add current to previous (working backwards)
		int result = sumOfSeriesRecursion(n - 1) + n;
		System.out.println("SumOfSeriesRecursion(" + n + ") = " + result);
		return result;
	}
}
```

Stopping condition.

Tracing the execution of sumOfSeriesRecursion(3)



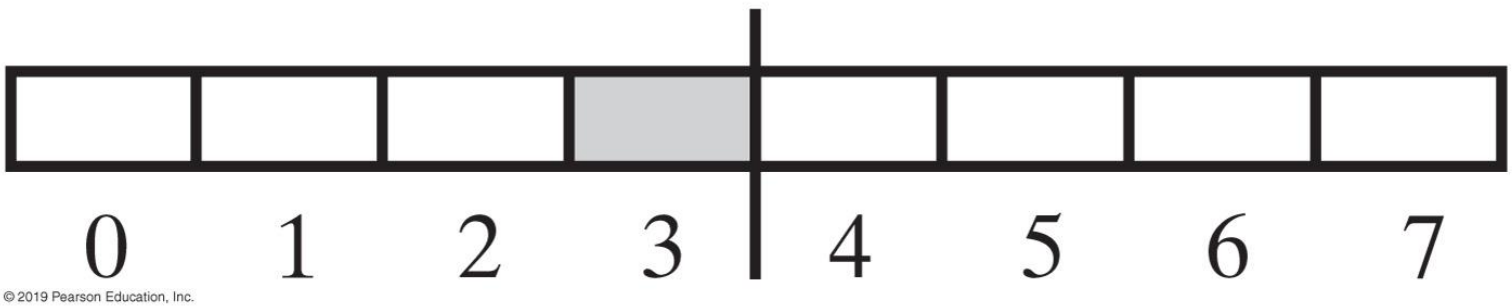
Recursively processing an array

```java
public class DisplayArrayUsingRecursionDemo {

    public static void main(String[] args) {
        String[] strings = { "a", "b", "c", "d", "e" };
        displayArray(strings, 2, 4);
        displayArrayFromMiddle(strings, 2, 4);

        Integer[] integers = { 5, 7, 9, 11, 16, 3, 4, 9 };
        displayArray(integers, 1, integers.length - 1);
        displayArrayFromMiddle(integers, 1, integers.length - 1);
    }

    /**
     * Displays the integers in an array.
     *
     * @param array An array of integers.
     * @param first The index of the first integer displayed.
     * @param last  The index of the last integer displayed, 0 <= first <= last <
     *              array.length.
     * @author Frank M. Carrano, Timothy M. Henry
     * @version 5.0
     *
     * @author mhrybyk
     *
     * Made it generic and added ending newline
     * @param <T>
     */
    public static <T> void displayArray(T[] array, int first, int last) {
        System.out.print(array[first] + " ");
        if (first < last)
            displayArray(array, first + 1, last);
        else
            System.out.println(); // we are at the end
    }
```

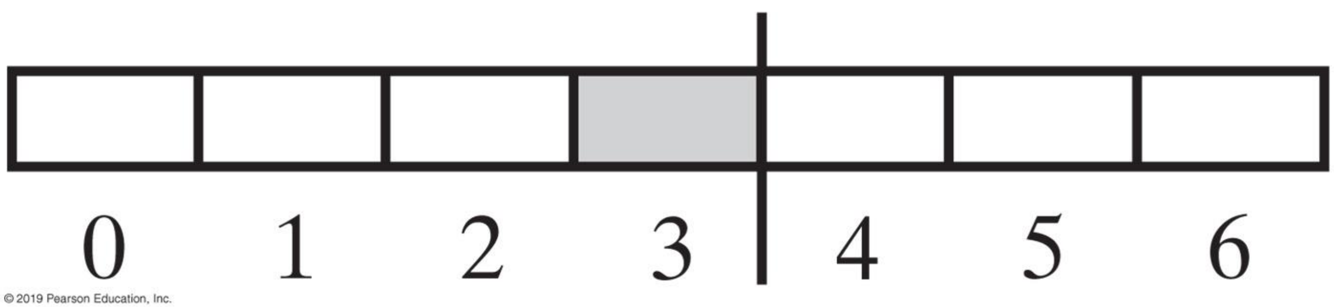
Recursively Processing an Array

- Two arrays with their middle elements within their left halves

(a)



(b)



Recursively processing an array

```
/**
 * Display an array from the middle using divide and conquer
 * @param array
 * @param first
 * @param last
 */
public static <T> void displayArrayFromMiddle(T[] array, int first, int last) {
     displayArrayFromMiddleHelper(array, first, last);
     System.out.println();
}

public static <T> void displayArrayFromMiddleHelper(T[] array, int first, int last) {
     if (first == last)
          System.out.print(array[first] + " ");
     else {
          int mid = first + ((last - first) / 2);
          displayArrayFromMiddleHelper(array, first, mid);
          displayArrayFromMiddleHelper(array, mid + 1, last);
     }
}
```

Traversing an LList recursively

- Forwards: display node data, then call recursively
- Backwards: call recursively, then display node data
    - call stack builds up, then pops as it unwinds
- See **DisplayListUsingRecursion**

```java
private static <T> void traverseWithIndex(ListInterface<T> list) {
     System.out.println("Traversing list with index forward");
     int i = 0;
     displayWithIndex(list, i);
     System.out.println();

}
private static <T> void displayWithIndex(ListInterface<T> list, int i) {
     if(i < list.size()) {
          T data = list.getEntry(i);  // get data
          System.out.print(data + " "); // display data
          displayWithIndex(list, i + 1); // Display rest of the list
     }
}
private static <T> void traverseWithIndexBackward(ListInterface<T> list) {
     System.out.println("Traversing list with index backward");
     int i = 0;
     displayWithIndexBackward(list, i);
     System.out.println();

}
private static <T> void displayWithIndexBackward(ListInterface<T> list, int i) {
     if(i < list.size()) {
          T data = list.getEntry(i);  // get data
          displayWithIndexBackward(list, i + 1); // Display rest of the list
          System.out.print(data + " "); // display data
     }
}
```

Traverse an LList recursively using Iterator

```
static public <T> void traverseWithIterator(ListWithIteratorInterface<T> list) {
     System.out.println("Traversing list forward");
     Iterator<T> iterator = list.getIterator();
     displayWithIterator(iterator);
     System.out.println();
}
static public <T> void displayWithIterator(Iterator<T> iterator) {
     if (iterator.hasNext()) {
          T data = iterator.next();  // get data
          System.out.print(data + " "); // display data
          displayWithIterator(iterator); // Display rest of the list
     }
}
static public <T> void traverseBackwardWithIterator(ListWithIteratorInterface<T> list) {
     System.out.println("Traversing list backward");
     Iterator<T> iterator = list.getIterator();
     displayWithIteratorBackward(iterator);
     System.out.println();
}
private static <T> void displayWithIteratorBackward(Iterator<T> iterator) {
     if (iterator.hasNext()) {
          T data = iterator.next();  // get data
          displayWithIteratorBackward(iterator); // display rest of the list
          System.out.print(data + " "); // display data
     }

}
```

Time Efficiency of Recursive Methods

- Using proof by induction, we conclude method is O(n).

```java
public static void countDown(int n)
{
  System.out.println(n);
  if (n > 1)
    countDown(n - 1);
} // end countDown
```

Recursive algorithm for a^b

- If b is even, $a^b = a^{b/2} * a^{b/2}$
- If b is odd, $a^b = a * a^{b/2} * a^{b/2}$
- Example a^6
  - $a^6 = a^3 * a^3$
  - $a^3 = a * a^1 * a^1$ (due to integer division and odd number)
  - Number of multiplies = 3
- Example 2^6
  - $2^6 = 2^3 * 2^3 = 64$
  - $2^3 = 2 * 2^1 * 2^1 = 8$

RaiseToAPower – recursive implementation

- See power() method in the example code

```
if (exponent == 0) {
    System.out.println(" base case returns 1");
    return 1;
}

// halve exponent and get the result
// we will square this later

int halfExponent = exponent / 2;
int halfResult = powerHelper(n, halfExponent);


// this will be the result returned to the caller
int result = 0;

// square the result if odd exponent, multiply by another copy of the base n

if (exponent % 2 == 1) {
    // if the result is > 1, two multiplies will be needed
    if (halfResult > 1)
        numberOfOperations += 2;
    result = n * halfResult * halfResult;

    return result;
} else {
    numberOfOperations++;
    result = halfResult * halfResult;
    return result;
}
```

Time Efficiency of Computing x^n

- Efficiency of algorithm is O(log *n*)
  - Because of divide and conquer

$x^n = (x^{n/2})^2$ when n is even and positive

$x^n = x\,(x^{(n-1)/2})^2$ when n is odd and positive

$x^0 = 1$

Other examples

- Greatest common divisor
  - Euclid method
    - Keep subtracting small from larger until the the positions are switched, then swap
  - Modulus method
    - Compute larger % smaller, then swap
    - Note this is the same as above!
- Findlargest and Findposition
  - Use divide and conquer
  - Split array, then recursively call on upper or lower half
- Palindrome
  - Can use a character stack to reverse letters then compare.
  - Or compare first and last and call recursively.

Tail Recursion

- When the last action performed by a recursive method is a recursive call.
- In a tail-recursive method, the last action is a recursive call
- This call performs a repetition that can be done by using iteration.
- Converting a tail-recursive method to an iterative one is usually a straightforward process.

```
public static void countDown(int n)
{
   System.out.println(n);
   if (n > 1)
      countDown(n - 1);
} // end countDown
```

Tail recursion is really iterative

- Converting a recursive method to an iterative one

```
public static void countDown(int integer)
{
  if (integer >= 1)
  {
    System.out.println(integer);
    countDown(integer - 1);
  } // end if
} // end countDown
```

- An iterative version

```
public static void countDown(int integer)
{
  while (integer >= 1)
  {
    System.out.println(integer);
    integer = integer – 1;
  } // end while
} // end countDown
```

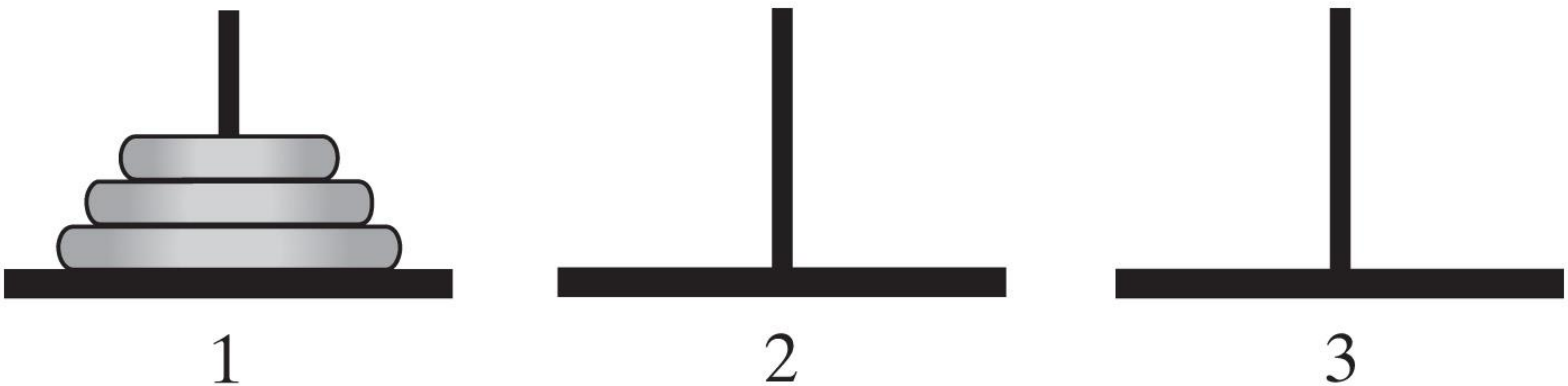
Tail recursion – sum of series

- Recursive call needs to be the last
- Usually requires a helper method

```java
/**
 * Compute sum of integer series using tail recursion
 *
 * @param n last term of integer series
 * @return sum of series
 */
static int sumOfSeriesTailRecursion(int n) {
	return sumOfSeriesTailRecursionHelper(n, 0);
}

/**
 * Helper method for computing sum of integer series using tail recursion
 * This is basically like iteration!
 *
 * @param n
 * @param sum partial result
 * @return
 */
static int sumOfSeriesTailRecursionHelper(int n, int sum) {
	int result = 0;
	System.out.println("SumOfSeriesTailRecursionHelper(" + n + ") partial sum = " + sum);

	// base case is whatever we have as sum and then add 1 to it
	if (n == 1)
		result = sum + 1;
	else {
		result = sumOfSeriesTailRecursionHelper(n - 1, sum + n);
	}

	System.out.println("SumOfSeriesTailRecursionHelper(" + n + ") = " + result);
	return result;
}
```

Towers of Hanoi

- The initial configuration of the Towers of Hanoi for three disks

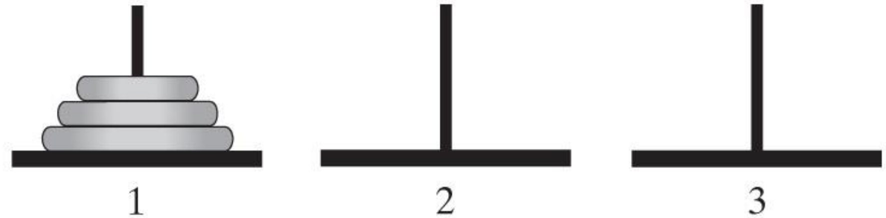


Towers of Hanoi

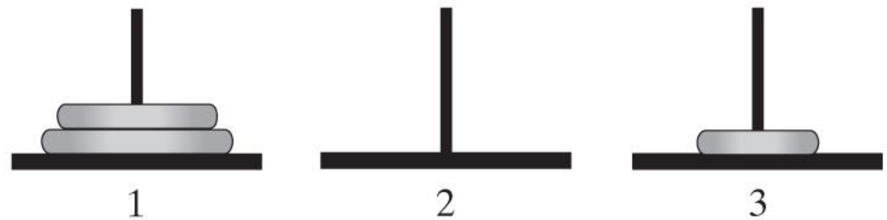
- Rules:
  - Move one disk at a time. Each disk moved must be the topmost disk.
  - No disk may rest on top of a disk smaller than itself.
  - You can store disks on the second (extra) pole temporarily, as long as you observe the previous two rules.
- See https://www.mathsisfun.com/games/towerofhanoi.html

Simple Solution to a Difficult Problem (Part 1)

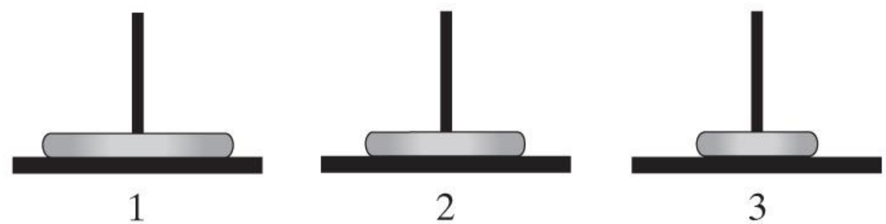
(a) The beginning configuration



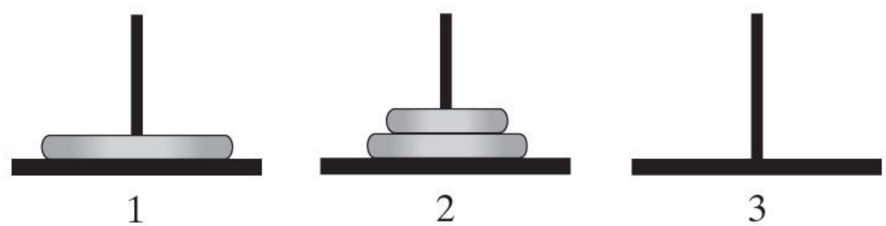
(b) After moving a disk from pole 1 to pole 3



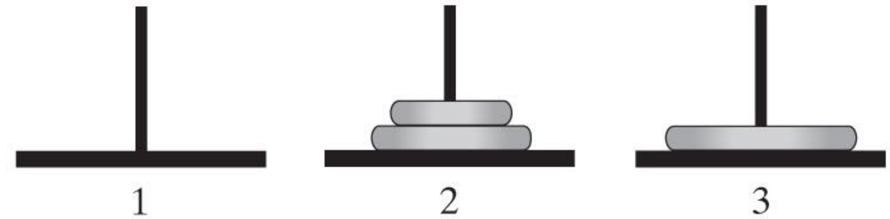
(c) After moving a disk from pole 1 to pole 2



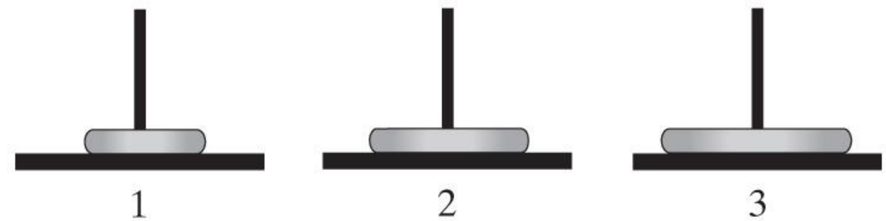
(d) After moving a disk from pole 3 to pole 2



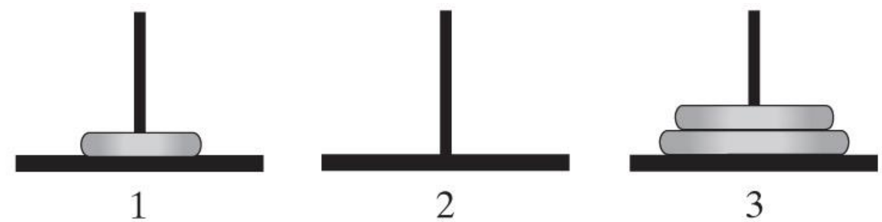
Simple Solution to a Difficult Problem (Part 2)



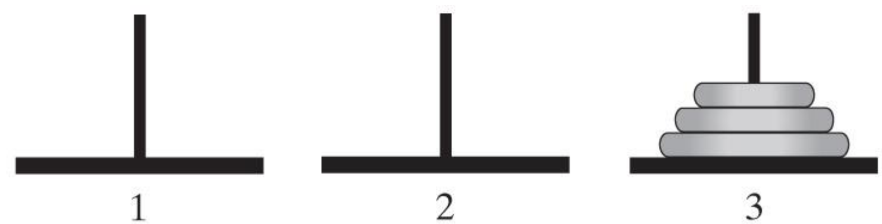
(e) After moving a disk from pole 1 to pole 3



(f) After moving a disk from pole 2 to pole 1

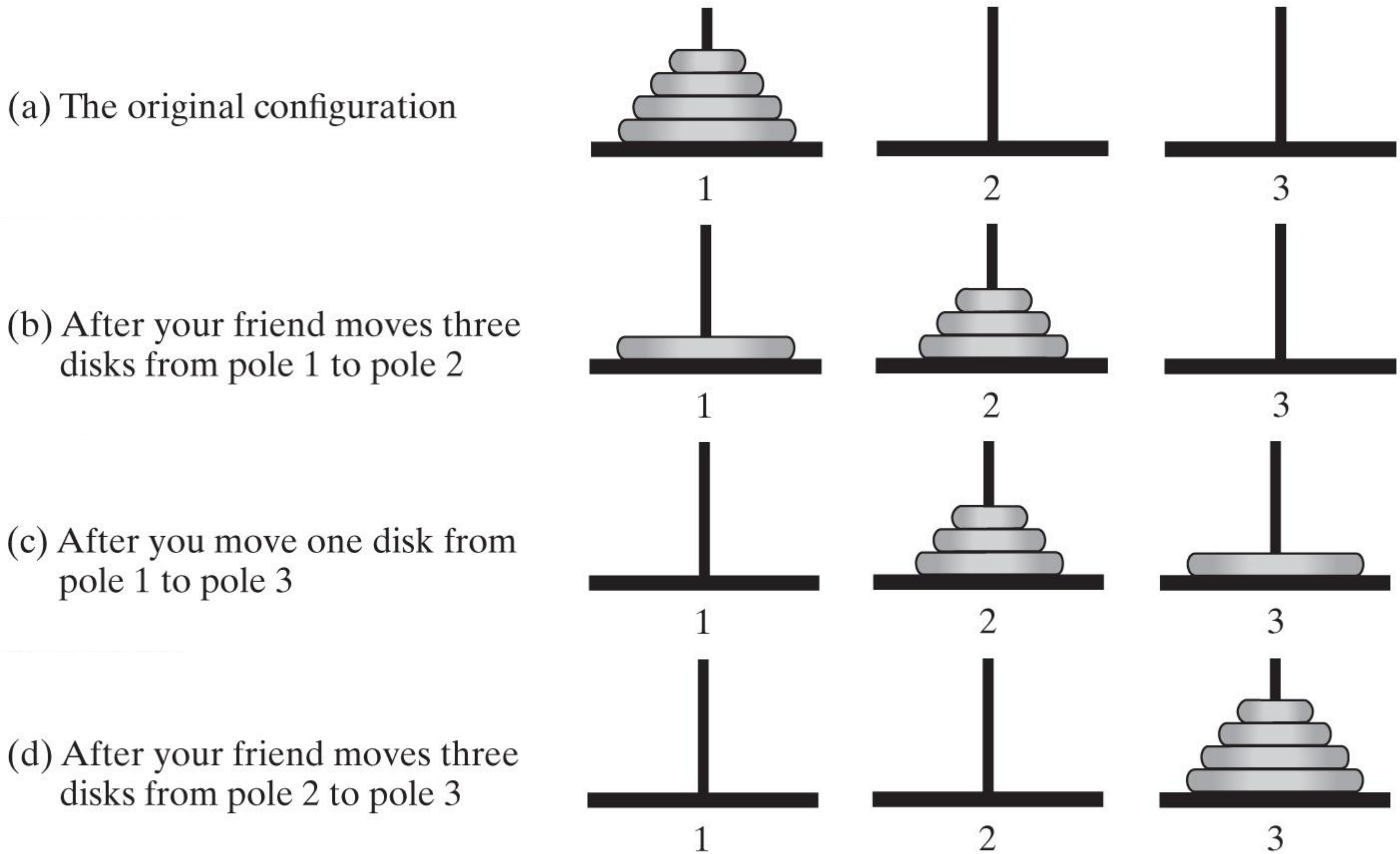


(g) After moving a disk from pole 2 to pole 3



(h) After moving a disk from pole 1 to pole 3

A Smaller Problem



Solutions

- Recursive algorithm to solve any number of disks.
  Note: for n disks, solution will be $2^n - 1$ moves

Algorithm to move* numberOfDisks *disks from* startPole *to* endPole *using* tempPole *as a spare according to the rules of the Towers of Hanoi problem

if (numberOfDisks == 1)

Move disk from startPole *to* endPole

else

{

Move all but the bottom disk from startPole *to* tempPole

Move disk from startPole *to* endPole

Move all disks from tempPole *to* endPole

}

Pseudo code for Towers of Hanoi

```
solveTowers(number of Disks, startPole, tempPole, endPole)

if (numberOfDisks == 1)
        Move disk from startPole to endPole
else {
        // Move all  but bottom from start to temp
        solveTowers(numberOfDisks – 1, startPole, endPole, tempPole)

        Move disk from startPole to EndPole

        // Move all disks from temp to end
        solveTowers(numberOfDisks – 1, tempPole, startPole, endPole)
}
```

Towers of Hanoi

```
    public static void main(String[] args) {

            // solve the tower of hanoi with a pre-fixed number of disks

        int n = 6; // Number of disks

        // move disks from A to C using B as temp
        // name the rods A, B, and C. Note that the third arg is the temp rod

        towerOfHanoi(n, 'A', 'C', 'B');
    }
    /**
     * Solve the towers of hanoi using recursion
     *
     * @param numberOfDisks number of disks
     * @param startRod starting rod
     * @param endRod ending rod
     * @param tempRod temp rod
     */
  static void towerOfHanoi(int numberOfDisks, char startRod, char endRod, char tempRod)
  {
      if (numberOfDisks == 1)
      {
          System.out.println("Move disk 1 from rod " +  startRod + " to rod " + endRod);
          return;
      }

      // move all but one disk from startRod to tempRod using endRod as temp
      towerOfHanoi(numberOfDisks - 1, startRod, tempRod, endRod);

      // move disk from startRod to endRod
      System.out.println("Move disk " + numberOfDisks + " from rod " +  startRod + " to rod " + endRod);

      // move all but one disk from tempRod to endRod using startRod as temp
      towerOfHanoi(numberOfDisks - 1, tempRod, endRod, startRod);
  }
```

Poor Solution to a Simple Problem

- Algorithm to generate Fibonacci numbers.
- Why is this inefficient?

$F_0 = 1$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$ when $n \geq 2$

Algorithm **Fibonacci(n) if** (n <= 1)

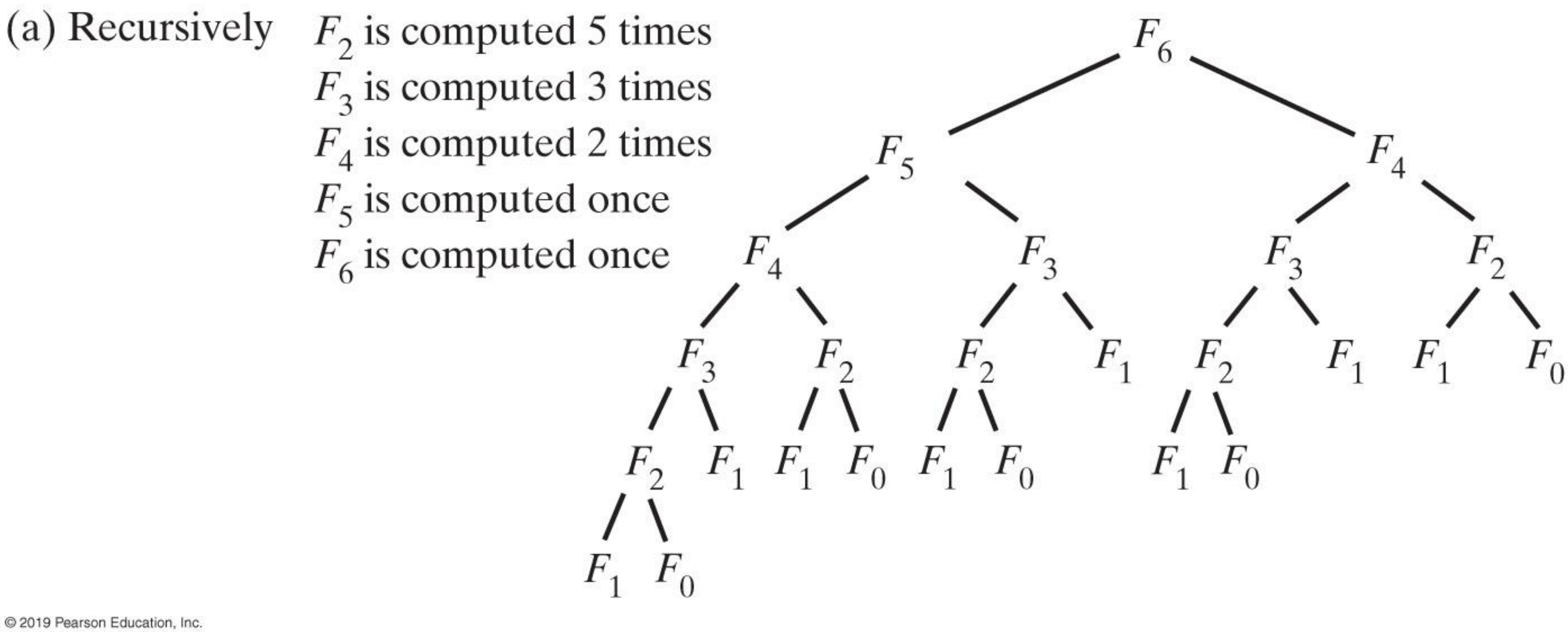
  return 1

else

  return Fibonacci(n − 1) + Fibonacci(n − 2)

Poor Solution to a Simple Problem

- The computation of the Fibonacci number F6



Poor Solution to a Simple Problem

- The computation of the Fibonacci number F6

(a) Recursively

F_2 is computed 5 times
F_3 is computed 3 times
F_4 is computed 2 times
F_5 is computed once
F_6 is computed once

(b) Iteratively

$F_0 = 1$
$F_1 = 1$
$F_2 = F_1 + F_0 = 2$
$F_3 = F_2 + F_1 = 3$
$F_4 = F_3 + F_2 = 5$
$F_5 = F_4 + F_3 = 8$
$F_6 = F_5 + F_4 = 13$

Indirect Recursion

- Example
  - Method A calls Method B
  - Method B calls Method C
  - Method C calls Method A

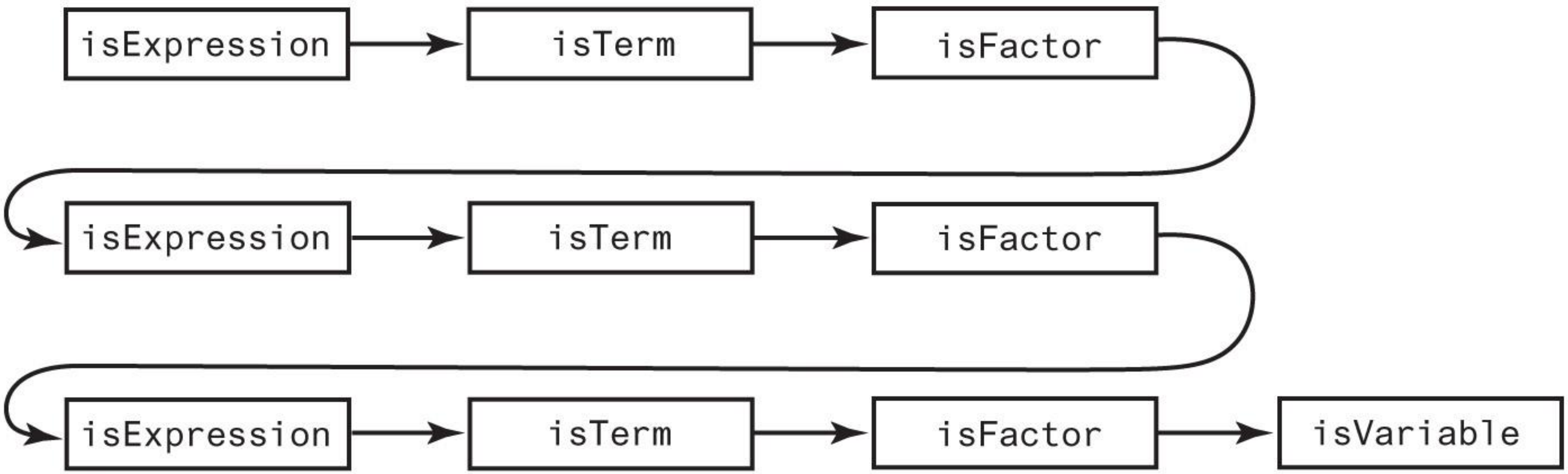
- Difficult to understand and trace
  - But does happen occasionally

Indirect Recursion

- Consider evaluation of validity of an algebraic expression
  - Algebraic expression is either a term or two terms separated by a + or – operator
  - Term is either a factor or two factors separated by a * or / operator
  - Factor is either a variable or an algebraic expression enclosed in parentheses
  - Variable is a single letter

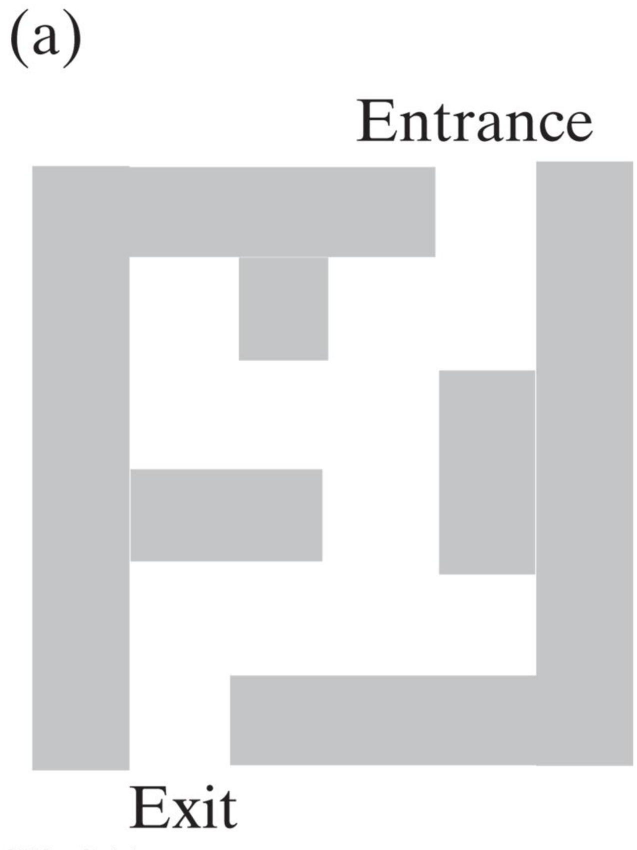
Indirect Recursion

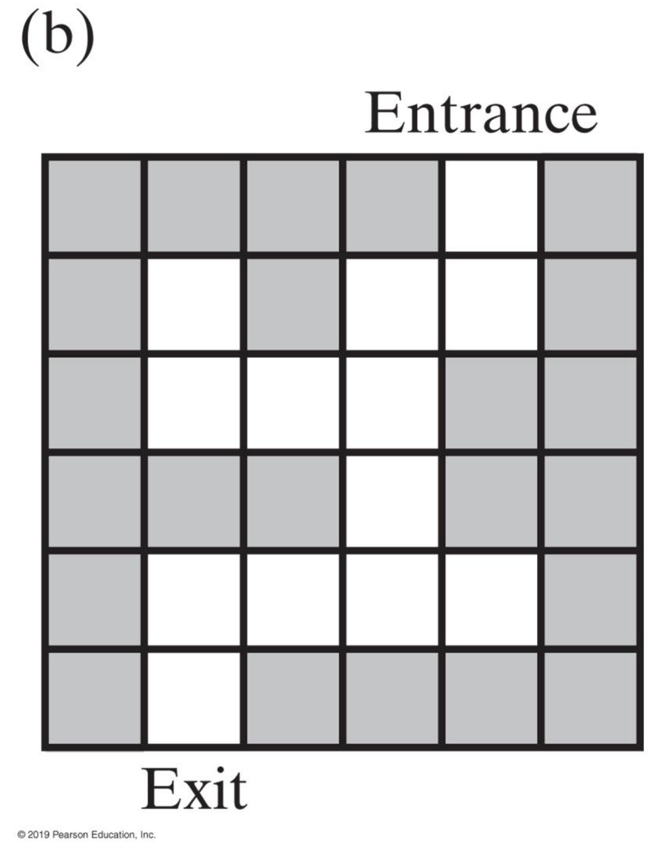
- FIGURE 14-5 An example of indirect recursion



Backtracking

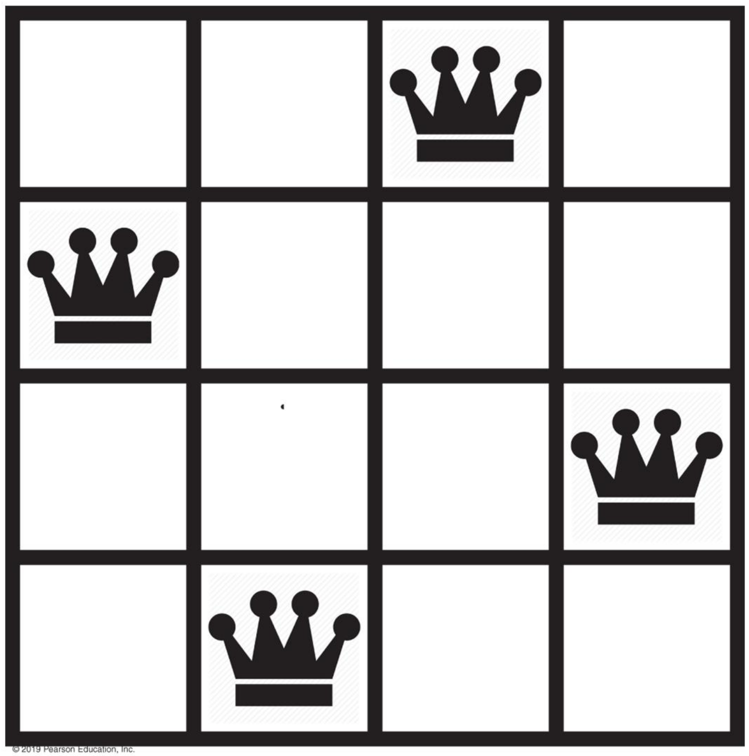
- A two-dimensional maze with one entrance and one exit
- Solve via backtracking





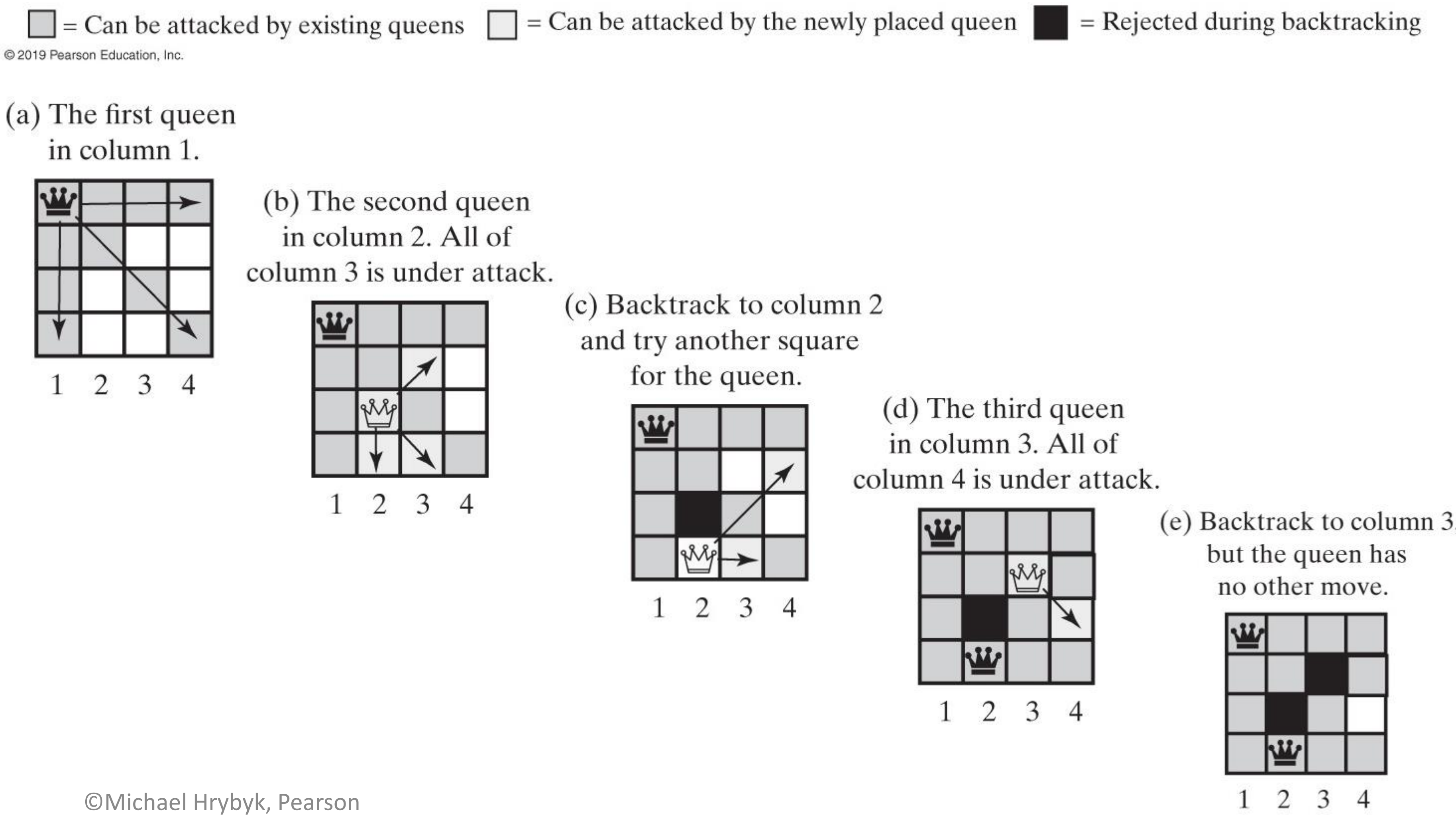
Backtracking

- A solution to the four-queens problem



Backtracking - Queens Solution (Part 1)

- Solving the four-queens problem by placing one queen at a time in each column



Backtracking - Queens Solution (Part 2)

- Solving the four-queens problem by placing one queen at a time in each column