

## Chapter 3

# Recursion: The Mirrors

# Recursive Solutions

- Recursion
  - An extremely powerful problem-solving technique
  - Breaks a problem in smaller identical problems
  - An alternative to iteration
    - An iterative solution involves loops

# Recursive Solutions

- Sequential search
  - Starts at the beginning of the collection
  - Looks at every item in the collection in order until the item being searched for is found
- Binary search
  - Repeatedly halves the collection and determines which half could contain the item
  - Uses a divide and conquer strategy

# Recursive Solutions

- Facts about a recursive solution
  - A recursive method calls itself
  - Each recursive call solves an identical, but smaller, problem
  - A test for the base case enables the recursive calls to stop
    - Base case: a known case in a recursive definition
  - Eventually, one of the smaller problems must be the base case

# Recursive Solutions

- Four questions for construction recursive solutions
  - How can you define the problem in terms of a smaller problem of the same type?
  - How does each recursive call diminish the size of the problem?
  - What instance of the problem can serve as the base case?
  - As the problem size diminishes, will you reach this base case?

# A Recursive Valued Method: The Factorial of n

- Problem
  - Compute the factorial of an integer n
- An iterative definition of factorial(n)

$$\text{factorial}(n) = n * (n-1) * (n-2) * \dots * 1$$

for any integer  $n > 0$

$$\text{factorial}(0) = 1$$

# A Recursive Valued Method: The Factorial of n

- A recursive definition of factorial(n)

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{if } n > 0 \end{cases}$$

- A recurrence relation
  - A mathematical formula that generates the terms in a sequence from previous terms
  - Example

$$\begin{aligned} \text{factorial}(n) &= n * [(n-1) * (n-2) * \dots * 1] \\ &= n * \text{factorial}(n-1) \end{aligned}$$

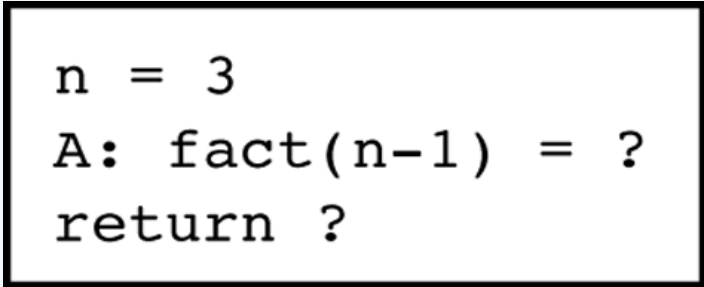
# A Recursive Valued Method: The Factorial of $n$

- Box trace
  - A systematic way to trace the actions of a recursive method
  - Each box roughly corresponds to an activation record
  - An activation record
    - Contains a method's local environment at the time of and as a result of the call to the method



# A Recursive Valued Method: The Factorial of n

- A method's local environment includes:
  - The method's local variables
  - A copy of the actual value arguments
  - A return address in the calling routine
  - The value of the method itself



```
n = 3  
A: fact(n-1) = ?  
return ?
```

Figure 3-3

A box

# A Recursive `void` Method: Writing a String Backward

- Problem
  - Given a string of characters, write it in reverse order
- Recursive solution
  - Each recursive step of the solution diminishes by 1 the length of the string to be written backward
  - Base case
    - Write the empty string backward

# A Recursive `void` Method: Writing a String Backward

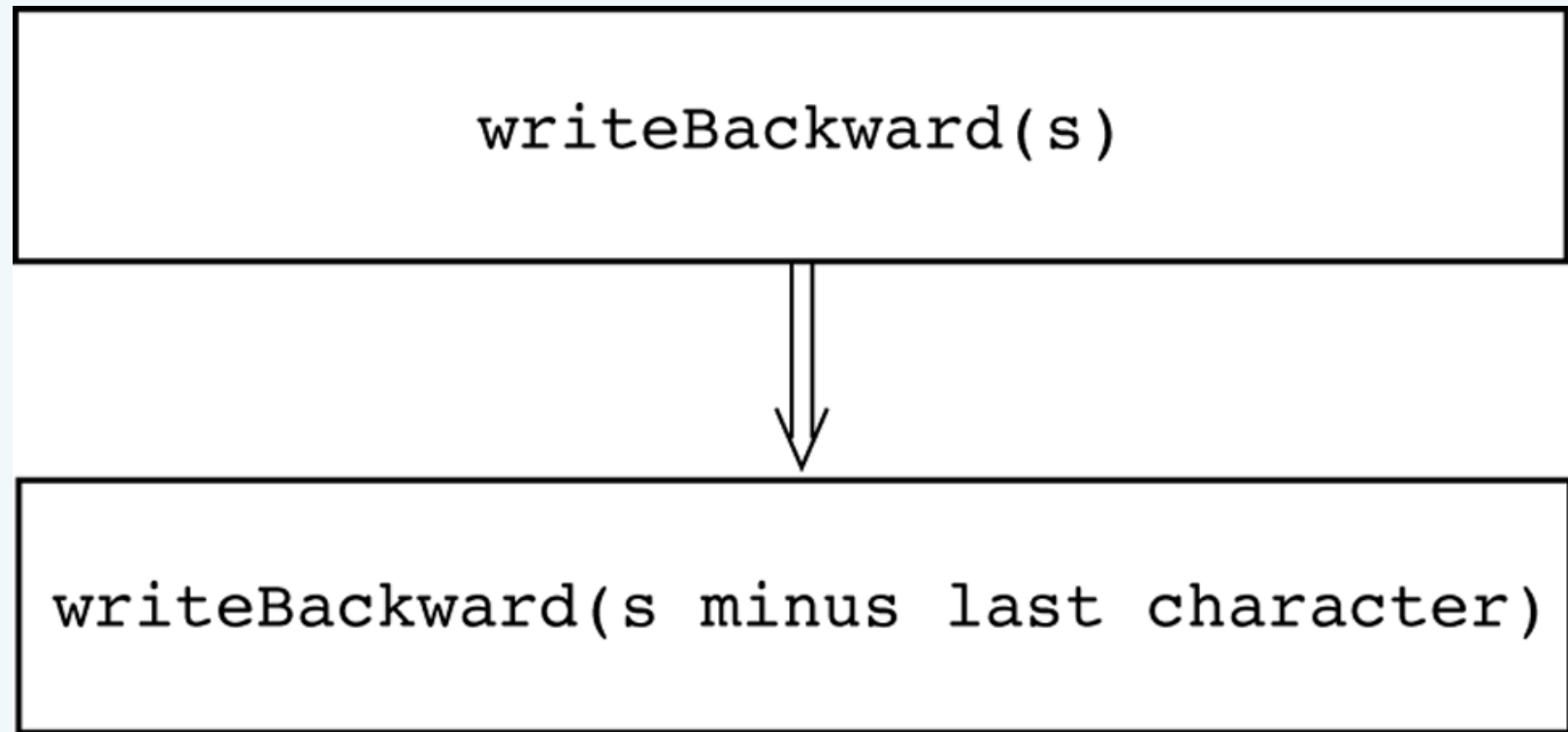


Figure 3-6

A recursive solution

# A Recursive `void` Method: Writing a String Backward

- Execution of `writeBackward` can be traced using the box trace
- Temporary `System.out.println` statements can be used to debug a recursive method

# Counting Things

- Next three problems
  - Require you to count certain events or combinations of events or things
  - Contain more than one base cases
  - Are good examples of inefficient recursive solutions

# Multiplying Rabbits (The Fibonacci Sequence)

- “Facts” about rabbits
  - Rabbits never die
  - A rabbit reaches sexual maturity exactly two months after birth, that is, at the beginning of its third month of life
  - Rabbits are always born in male-female pairs
    - At the beginning of every month, each sexually mature male-female pair gives birth to exactly one male-female pair

# Multiplying Rabbits (The Fibonacci Sequence)

- Problem
  - How many pairs of rabbits are alive in month  $n$ ?
- Recurrence relation

$$\text{rabbit}(n) = \text{rabbit}(n-1) + \text{rabbit}(n-2)$$

# Multiplying Rabbits (The Fibonacci Sequence)

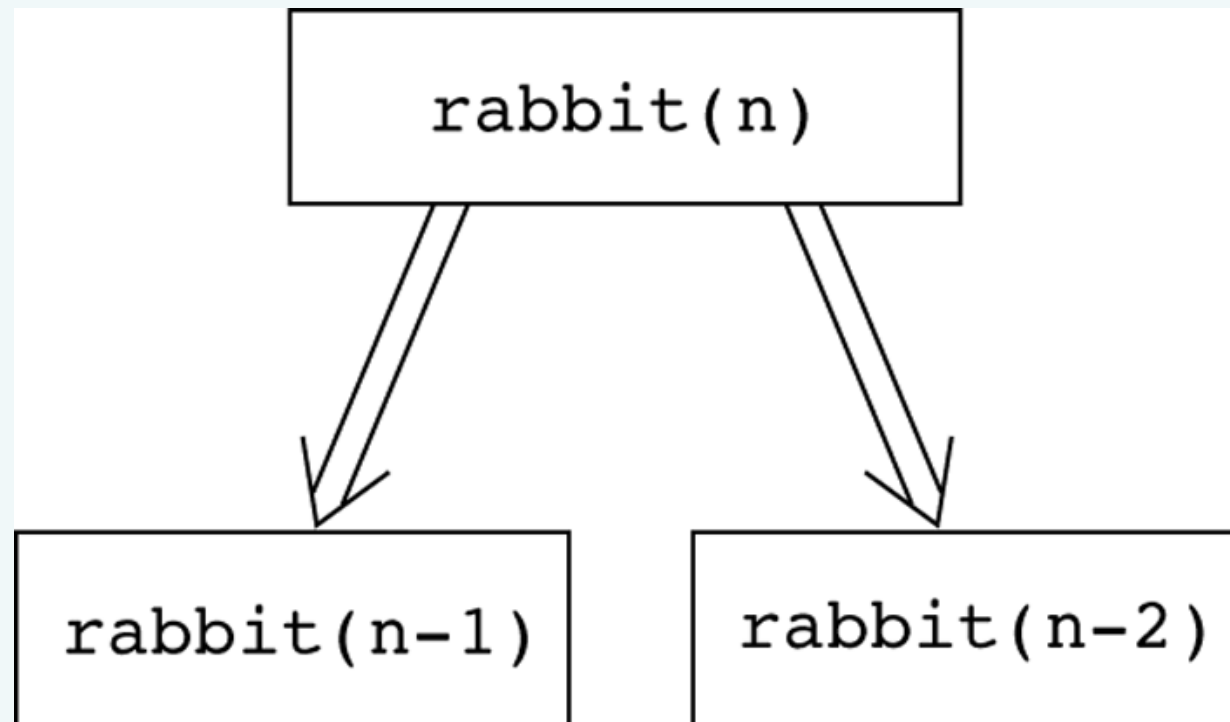


Figure 3-10

Recursive solution to the rabbit problem



# Multiplying Rabbits (The Fibonacci Sequence)

- Base cases
  - rabbit(2), rabbit(1)
- Recursive definition

$$\text{rabbit}(n) = \begin{cases} 1 & \text{if } n \text{ is 1 or 2} \\ \text{rabbit}(n-1) + \text{rabbit}(n-2) & \text{if } n > 2 \end{cases}$$

- Fibonacci sequence
  - The series of numbers rabbit(1), rabbit(2), rabbit(3), and so on

# Organizing a Parade

- Rules about organizing a parade
  - The parade will consist of bands and floats in a single line
  - One band cannot be placed immediately after another
- Problem
  - How many ways can you organize a parade of length  $n$ ?

# Organizing a Parade

- Let:
  - $P(n)$  be the number of ways to organize a parade of length  $n$
  - $F(n)$  be the number of parades of length  $n$  that end with a float
  - $B(n)$  be the number of parades of length  $n$  that end with a band
- Then
  - $P(n) = F(n) + B(n)$

# Organizing a Parade

- Number of acceptable parades of length  $n$  that end with a float

$$F(n) = P(n-1)$$

- Number of acceptable parades of length  $n$  that end with a band

$$B(n) = F(n-1)$$

- Number of acceptable parades of length  $n$ 
  - $P(n) = P(n-1) + P(n-2)$

# Organizing a Parade

- Base cases

$P(1) = 2$  (The parades of length 1 are float and band.)

$P(2) = 3$  (The parades of length 2 are float-float, band-float, and float-band.)

- Solution

$$P(1) = 2$$

$$P(2) = 3$$

$$P(n) = P(n-1) + P(n-2) \quad \text{for } n > 2$$

# Mr. Spock's Dilemma (Choosing $k$ out of $n$ Things)

- Problem
  - How many different choices are possible for exploring  $k$  planets out of  $n$  planets in a solar system?
- Let
  - $c(n, k)$  be the number of groups of  $k$  planets chosen from  $n$

# Mr. Spock's Dilemma (Choosing k out of n Things)

- In terms of Planet X:

$c(n, k)$  = (the number of groups of k planets that  
include Planet X)

+

(the number of groups of k planets that  
do not include Planet X)

# Mr. Spock's Dilemma (Choosing k out of n Things)

- The number of ways to choose k out of n things is the sum of
  - The number of ways to choose k-1 out of n-1 things and
  - The number of ways to choose k out of n-1 things

$$c(n, k) = c(n-1, k-1) + c(n-1, k)$$



# Mr. Spock's Dilemma (Choosing $k$ out of $n$ Things)

- Base cases
  - There is one group of everything  
 $c(k, k) = 1$
  - There is one group of nothing  
 $c(n, 0) = 1$
  - $c(n, k) = 0$       if  $k > n$

# Mr. Spock's Dilemma (Choosing k out of n Things)

- Recursive solution

$$c(n, k) = \begin{cases} 1 & \text{if } k = 0 \\ 1 & \text{if } k = n \\ 0 & \text{if } k > n \\ c(n-1, k-1) + c(n-1, k) & \text{if } 0 < k < n \end{cases}$$

# Searching an Array: Finding the Largest Item in an Array

- A recursive solution

```
if (anArray has only one item) {  
    maxArray(anArray) is the item in anArray  
}  
else if (anArray has more than one item) {  
    maxArray(anArray) is the maximum of  
        maxArray(left half of anArray) and  
        maxArray(right half of anArray)  
} // end if
```

# Finding the Largest Item in an Array

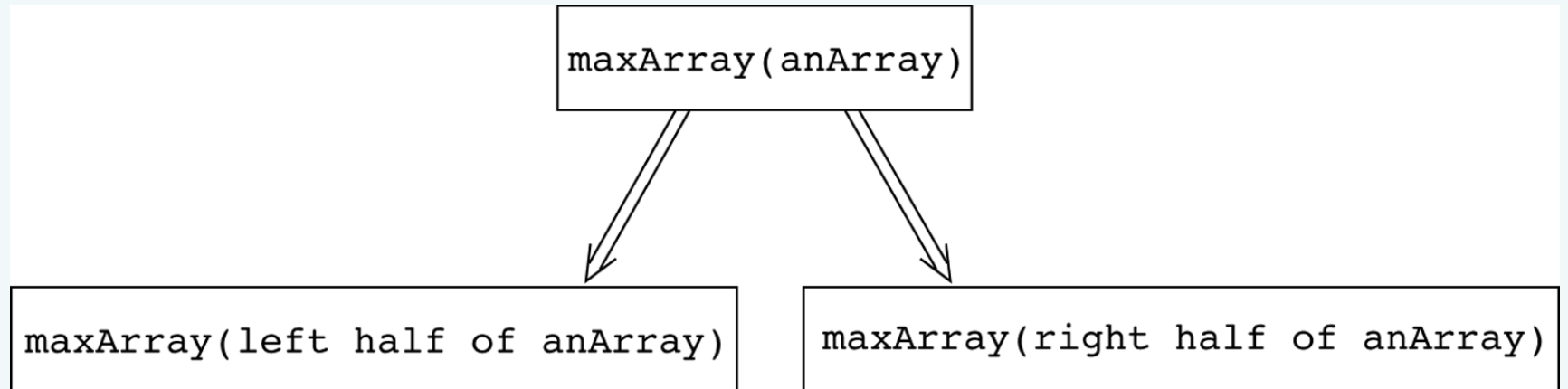


Figure 3-13

Recursive solution to the largest-item problem

# Binary Search

- A high-level binary search

```
if (anArray is of size 1) {  
    Determine if anArray's item is equal to value  
}  
else {  
    Find the midpoint of anArray  
    Determine which half of anArray contains value  
    if (value is in the first half of anArray) {  
        binarySearch (first half of anArray, value)  
    }  
    else {  
        binarySearch(second half of anArray, value)  
    } // end if  
} // end if
```

# Binary Search

- Implementation issues:
  - How will you pass “half of anArray” to the recursive calls to `binarySearch`?
  - How do you determine which half of the array contains value?
  - What should the base case(s) be?
  - How will `binarySearch` indicate the result of the search?

# Finding the $k^{\text{th}}$ Smallest Item in an Array

- The recursive solution proceeds by:
  1. Selecting a pivot item in the array
  2. Cleverly arranging, or partitioning, the items in the array about this pivot item
  3. Recursively applying the strategy to one of the partitions

# Finding the $k^{\text{th}}$ Smallest Item in an Array

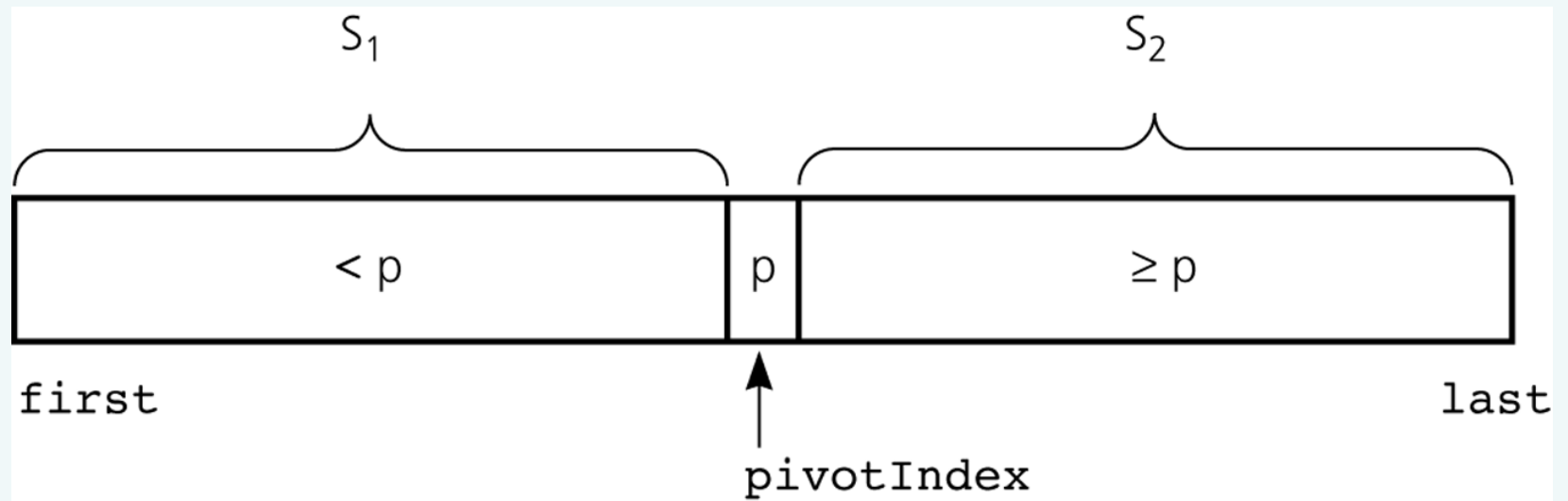


Figure 3-18

A partition about a pivot



# Finding the $k^{\text{th}}$ Smallest Item in an Array

- Let:

`kSmall(k, anArray, first, last) =`  
     $k^{\text{th}}$  smallest item in `anArray[first..last]`

- Solution:

`kSmall(k, anArray, first, last)`  
=  $\left\{ \begin{array}{l} \text{kSmall(k, anArray, first, pivotIndex-1)} \\ \quad \text{if } k < \text{pivotIndex} - \text{first} + 1 \\ \quad \text{if } k = \text{pivotIndex} - \text{first} + 1 \\ p \\ \text{kSmall(k-(pivotIndex-first+1), anArray,} \\ \quad \text{pivotIndex+1, last)} \\ \quad \text{if } k > \text{pivotIndex} - \text{first} + 1 \end{array} \right.$

# Organizing Data: The Towers of Hanoi

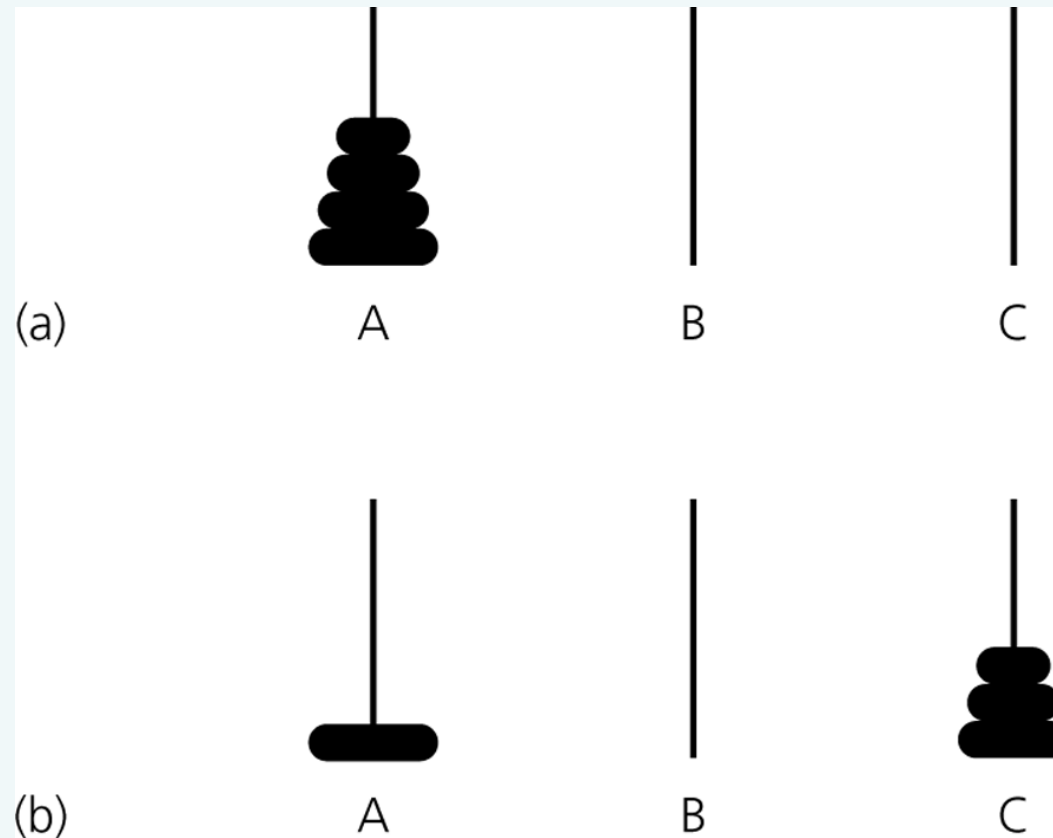


Figure 3-19a and b

a) The initial state; b) move  $n - 1$  disks from A to C

# The Towers of Hanoi

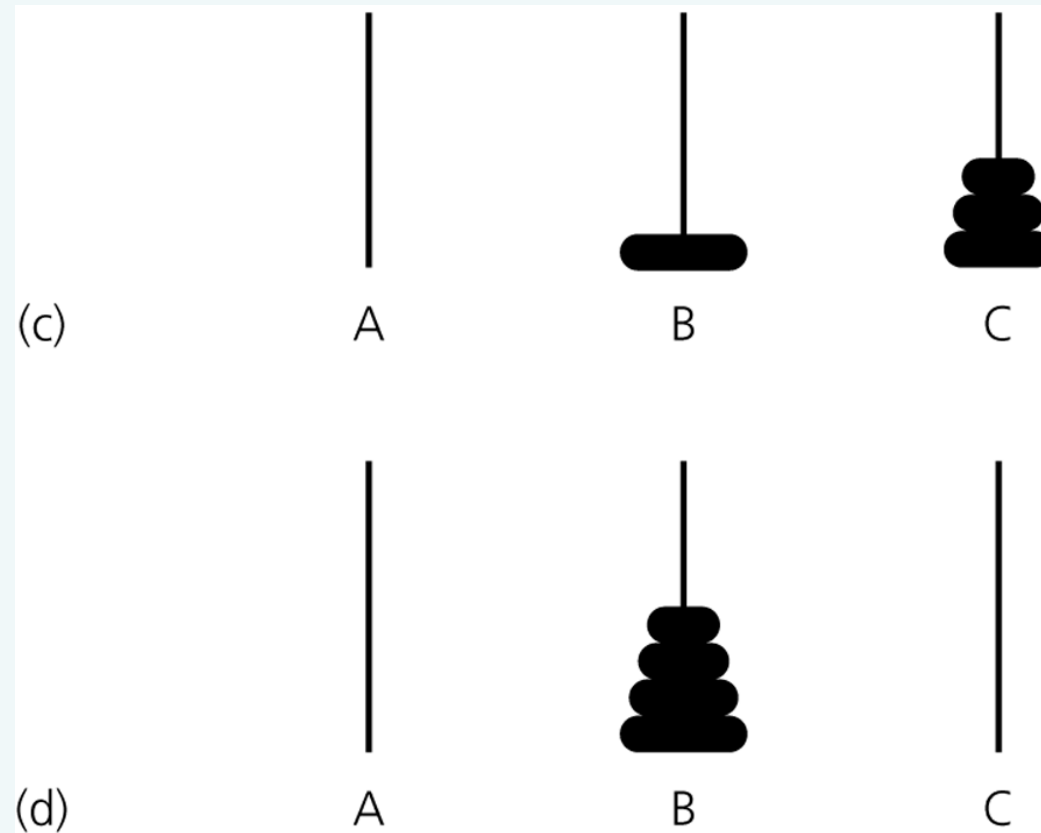


Figure 3-19c and d

c) move one disk from *A* to *B*; d) move  $n - 1$  disks from *C* to *B*

# The Towers of Hanoi

- Pseudocode solution

```
solveTowers(count, source, destination, spare)
  if (count is 1) {
    Move a disk directly from source to destination
  }
  else {
    solveTowers(count-1, source, spare, destination)
    solveTowers(1, source, destination, spare)
    solveTowers(count-1, spare, destination, source)
  } //end if
```

# Recursion and Efficiency

- Some recursive solutions are so inefficient that they should not be used
- Factors that contribute to the inefficiency of some recursive solutions
  - Overhead associated with method calls
  - Inherent inefficiency of some recursive algorithms

# Summary

- Recursion solves a problem by solving a smaller problem of the same type
- Four questions to keep in mind when constructing a recursive solution
  - How can you define the problem in terms of a smaller problem of the same type?
  - How does each recursive call diminish the size of the problem?
  - What instance of the problem can serve as the base case?
  - As the problem size diminishes, will you reach this base case?

# Summary

- A recursive call's postcondition can be assumed to be true if its precondition is true
- The box trace can be used to trace the actions of a recursive method
- Recursion can be used to solve problems whose iterative solutions are difficult to conceptualize

# Summary

- Some recursive solutions are much less efficient than a corresponding iterative solution due to their inherently inefficient algorithms and the overhead of method calls
- If you can easily, clearly, and efficiently solve a problem by using iteration, you should do so