# Chapter 2: Recursion

- Properties of recursive solutions
- Examples
- Efficiency

# Recursive Solutions

- Recursion is a programming pattern
  - function calls itself (on certain conditions)
- Solutions to some computing problems lend themselves naturally to recursion
  - solution is clearer
- Is a powerful problem-solving technique
  - breaks problem into smaller identical problems
  - alternative to iteration, which involves loops

# Recursive Solutions

- Facts about a recursive solution
  - a recursive function calls itself
  - each recursive call solves an identical, but smaller, problem
  - the solution to at least one smaller problem— the base case—is known
  - eventually, one of the smaller problems must be the base case; reaching the base case enables the recursive calls to stop!

# Recursive Solutions

- Four questions for constructing 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?

# Recursion Details

- Each function call (recursive or otherwise) pushes a new record on the *runtime* stack
  - contains arguments, locals, etc.
  - maintains function state
  - record popped on function return
  - introduces time and space overhead
- Box trace is visualize recursive call stack

# Box Trace

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

# A1: A Recursive Valued Function: The Factorial of n

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

  *factorial*(*n*) = *n* * (*n* – 1) * (*n* – 2) * … * 1

  for any integer *n* > 0

  *factorial*(0) = 1

- A recursive definition of *factorial*(*n*)

  *factorial*(*n*) = 1 if *n* = 0

  = *n* * *factorial*(*n*–1) if *n* > 0

*see C2-factorial.cpp*

# Box Trace

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

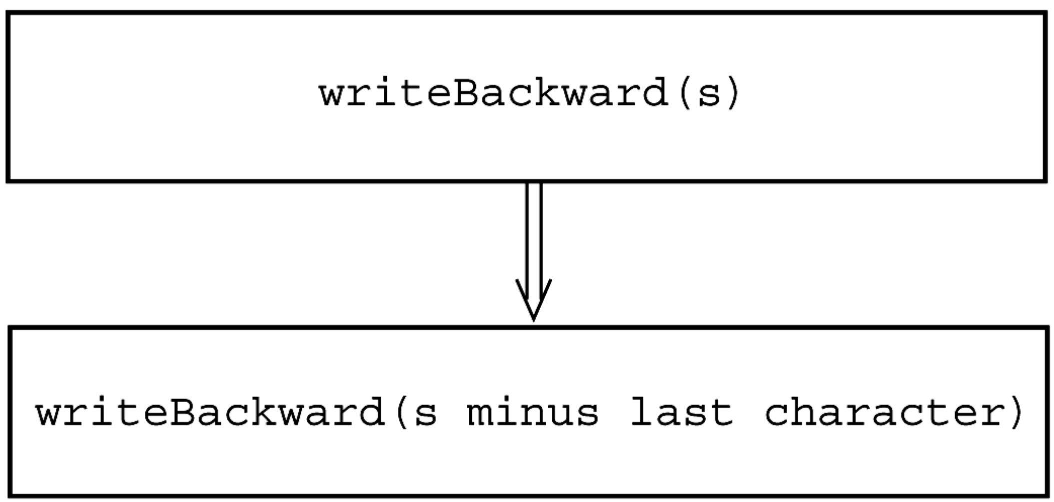
```
n = 3
A: fact(n-1) = ?
return ?
```

# A2: A Recursive void Function: 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

# A2: A Recursive void Function: Writing a String Backward

- Execution of writeBackward can be traced using the box trace
- Temporary cout statements can be used to debug a recursive method




see C2-backward.cpp

# A3: Fibonacci Sequence – Multiplying Rabbits

- Problem statement about rabbit growth
  - 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
  - How many pairs of rabbits are alive in month *n*?

# A3: Fibonacci Sequence – Multiplying Rabbits

- Recurrence relation

  rabbit(n) = rabbit(n – 1) + rabbit(n – 2)

- Base cases

  rabbit(2) = rabbit(1) = 1

- Recursive definition

  rabbit(n) = 1     If n is 1 or 2

  = rabbit(n – 1) + rabbit(n – 2)     if n > 2

- Fibonacci sequence
  - The series of numbers rabbit(1), rabbit(2), rabbit(3), and so on; that is, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

*see C2-fibonacci.cpp*

# A4: Organizing a Parade

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

# A4: 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
  - $P(n) = F(n) + B(n)$

# A4: 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
  - $P(n) = P(n-1) + P(n-2)$

# A4: 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

  $P(1) = 2$

  $P(2) = 3$

  $P(n) = P(n - 1) + P(n - 2)$ for $n > 2$

# A5: Choosing k out of n Things

- Problem statement
  - How many different choices are possible for exploring k planets out of n planets in a system?

# A5: Choosing k out of n Things

- Let c(n, k) be the number of groups of k planets chosen from n
- 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)
- Num. of ways to choose k of n things is the sum
  - the number of ways to choose k – 1 out of n – 1 things
  - the number of ways to choose k out of n – 1 things
  - $c(n, k) = c(n-1, k-1) + c(n-1, k)$

# A5: 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
  - Although k cannot exceed n here, we want our solution to be general

    c(n, k) = 0 if k > n
- Recursive solution

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

*see C2-kOfN.cpp*

# A5: Choosing k out of n Things

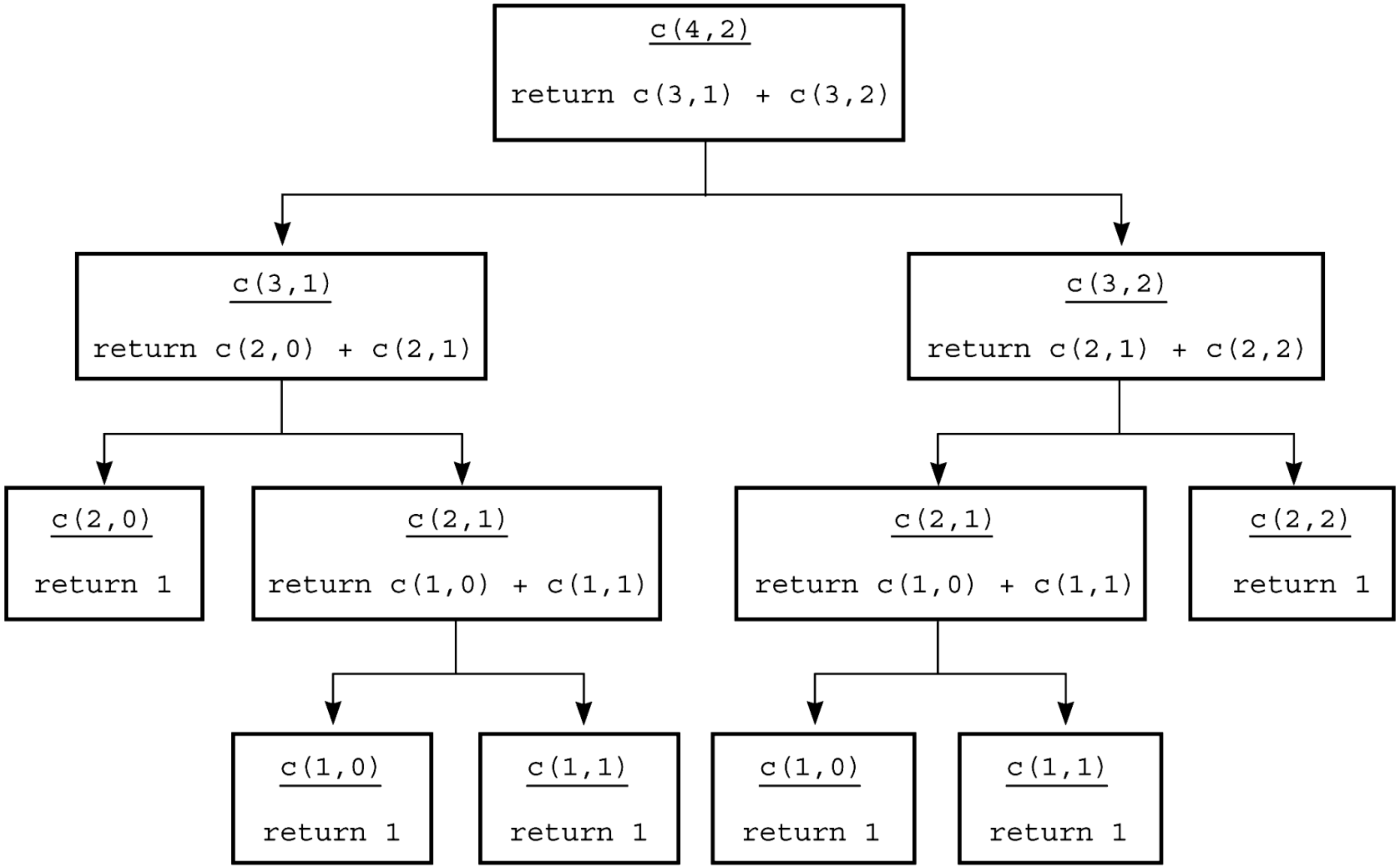



***Figure 2-12*** The recursive calls that *c(4, 2)* generates

# A6: Finding Largest Item in an Array

- A recursive solution – *maxArray*()

*if (anArray has only one item)*

*maxArray(anArray) is the item in anArray*

*else if (anArray has more than one item)*

*maxArray(anArray) is*

*MAX(maxArray(left half of anArray),*

*maxArray(right half of anArray) )*

# A7: Binary Search

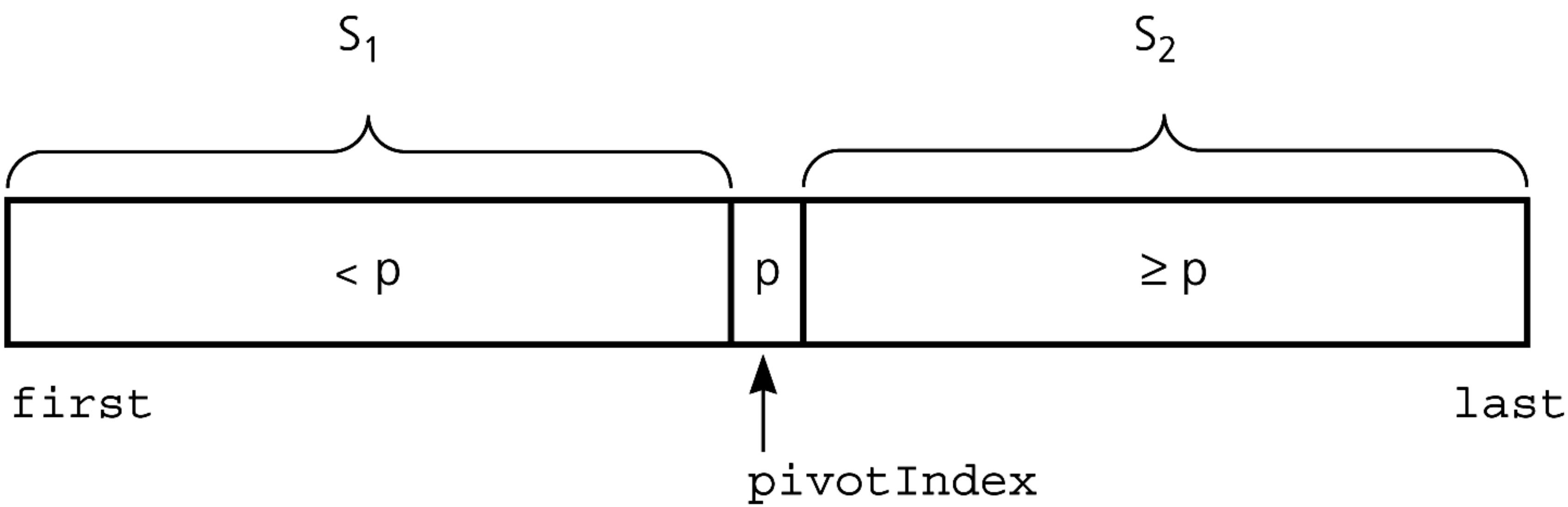
```
binarySearch(in anArray:ArrayType, in value:ItemType)

    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)
    }
```

*see C2-binarySearch.cpp*

# A8: Finding k$^{th}$ Smallest Item in Array

- Recursive solution
  - select a ‘pivot’ item in the array
  - partitioning items in array about this pivot item
  - recursively apply strategy to one of the partitions



# A8: Finding kth Smallest Item in Array

```
kSmall(k, anArray, first, last)
= kSmall(k,anArray,first,pivotIndex-1)
        if k < pivotIndex – first + 1
= p     if k = pivotIndex – first + 1
= kSmall(k-(pivotIndex-first+1), anArray,
         pivotIndex+1, last)
        if k > pivotIndex – first + 1
```

# A9: Organizing Data: The Towers of Hanoi

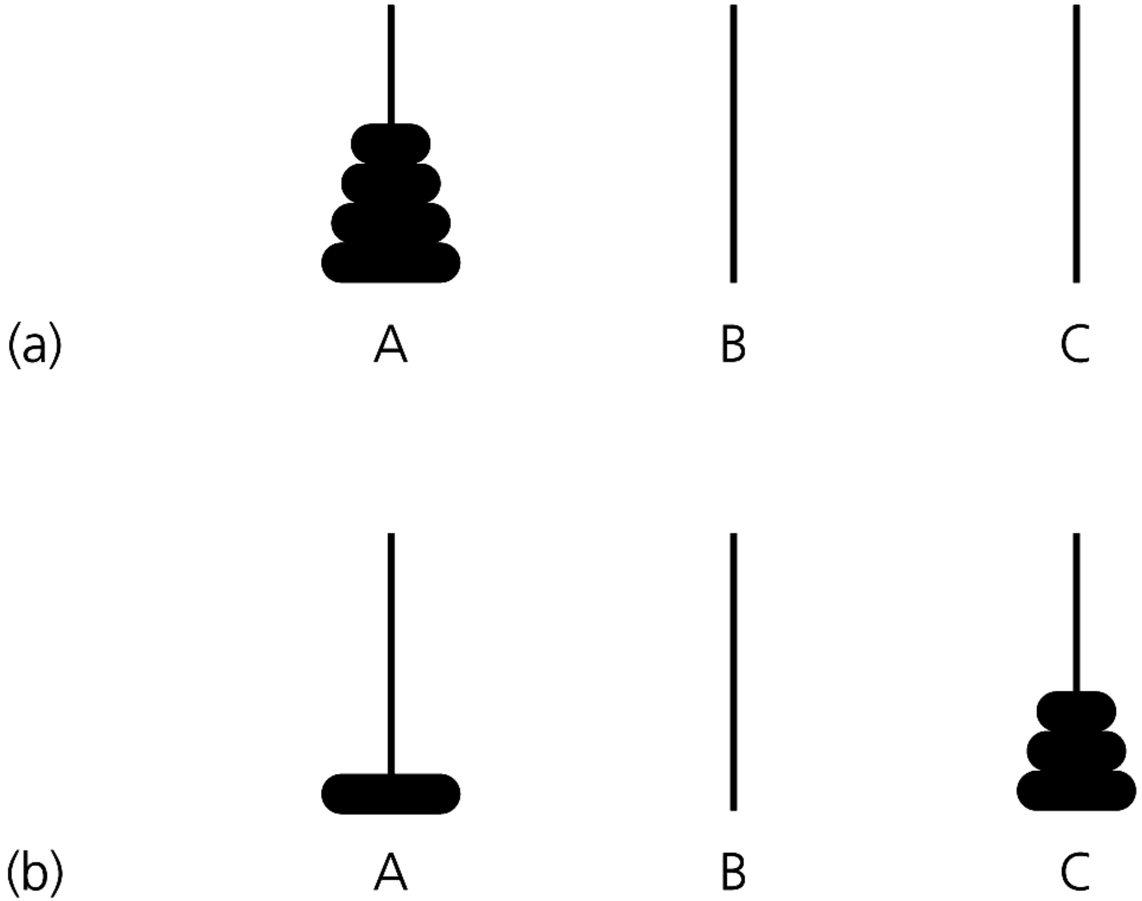



***Figure 2-19a and b*** (a) The initial state; (b) move *n* - 1 disks from *A* to *C*

# A9: The Towers of Hanoi

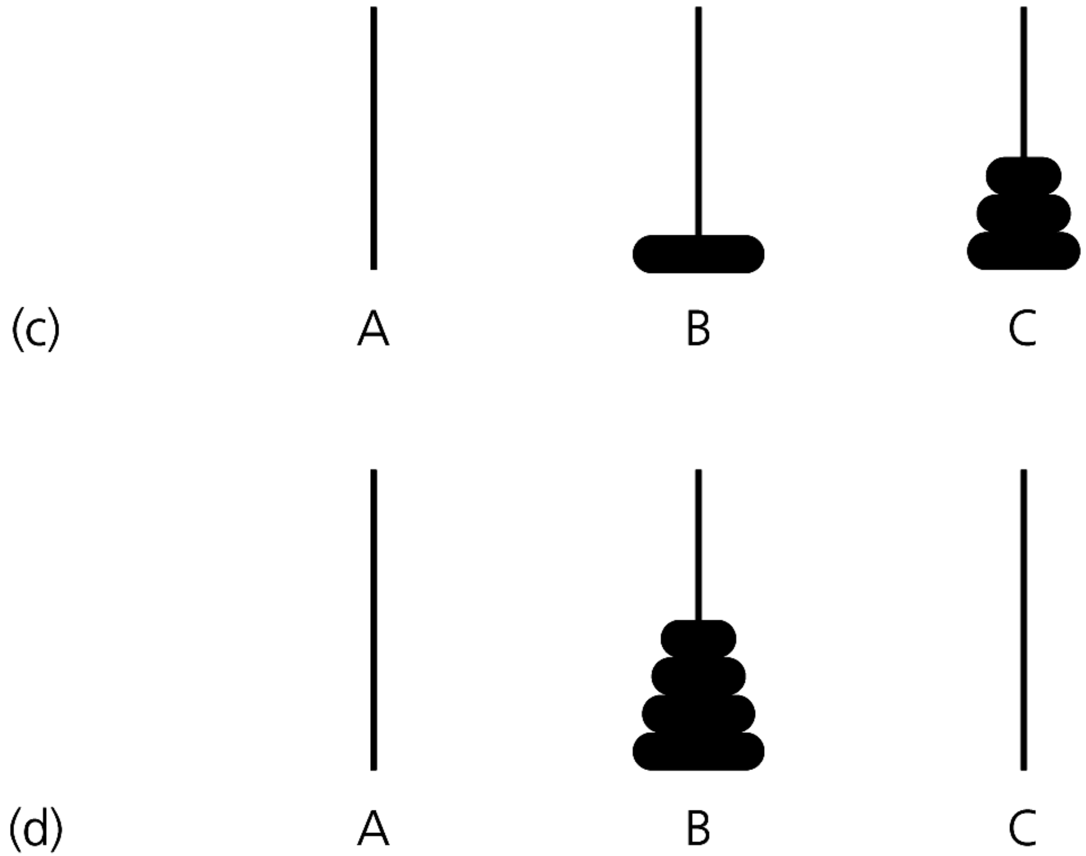



***Figure 2-19c and d*** (c) move one disk from *A* to *B*; (d) move *n* - 1 disks from *C* to *B*

# The Towers of Hanoi

```
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 function calls
  - inherent inefficiency of some recursive algorithms
- Do not use a recursive solution if it is inefficient and there is a clear, efficient iterative solution

# Summary

- Recursion solves a problem by solving a smaller problem of the same type
- Four questions:
  - 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(s) of the problem can serve as the base case?
  - As the problem size diminishes, will you reach a base case?

# Summary

- To construct a recursive solution, assume a recursive call's postcondition is 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 function calls
- If you can easily, clearly, and efficiently solve a problem by using iteration, you should do so