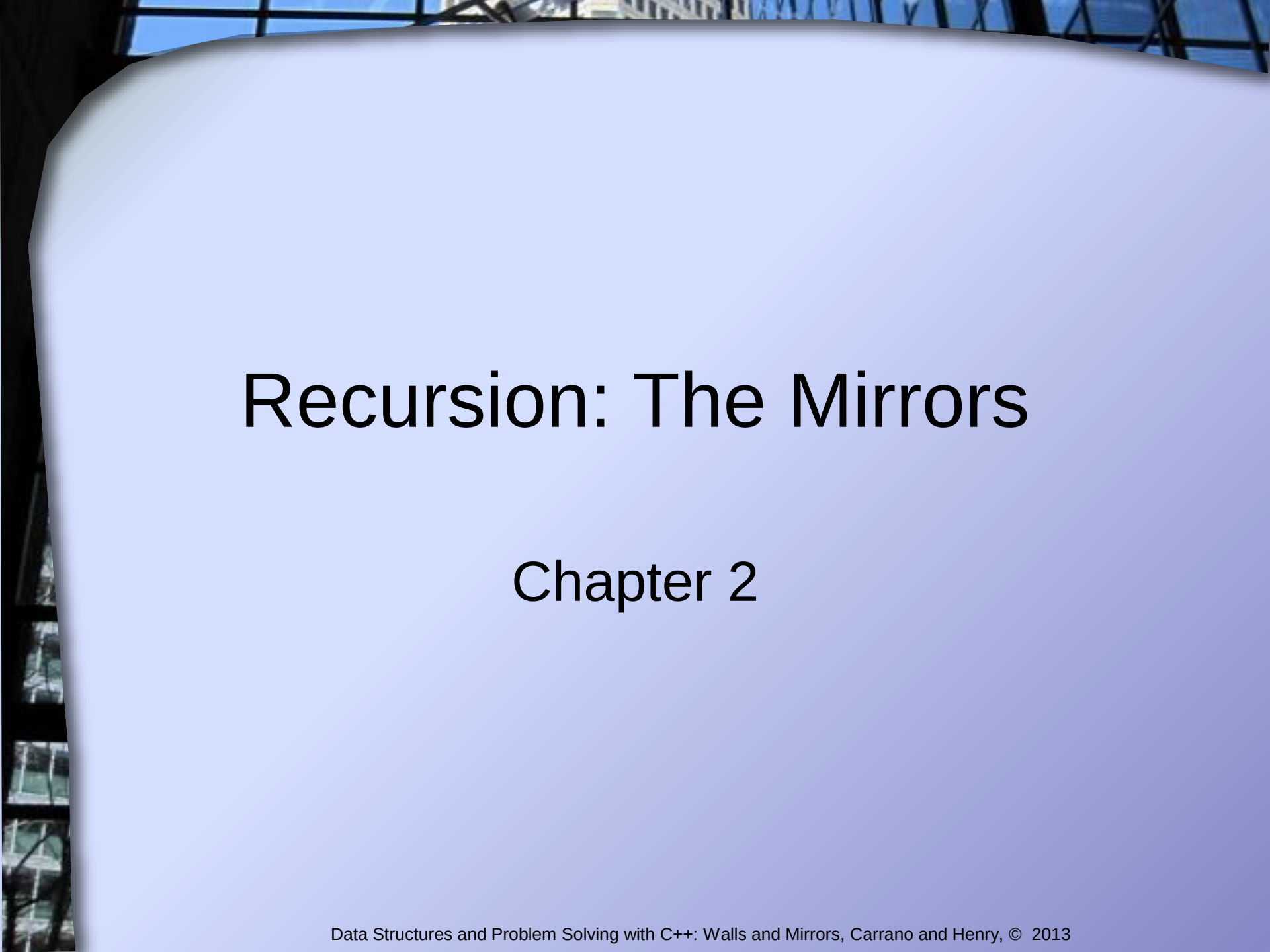


Recursion: The Mirrors

Chapter 2

Contents

- Recursive Solutions
- Recursion That Returns a Value
- Recursion That Performs an Action
- Recursion with Arrays
- Organizing Data
- More Examples
- Recursion and Efficiency

Recursive Solutions

- Recursion breaks a problem into smaller identical problems
- Some recursive solutions are inefficient, impractical
- Complex problems can have simple recursive solutions

Recursive Solutions

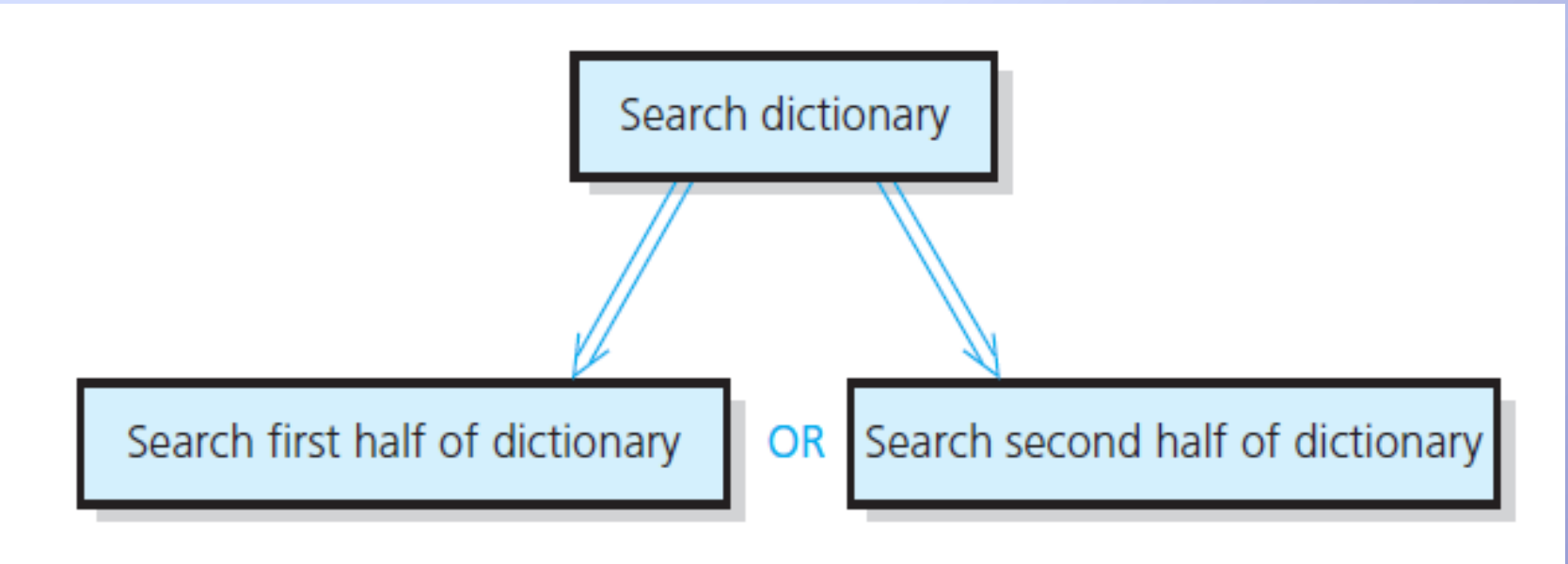


FIGURE 2-1 A recursive solution

Recursive Solutions

- A recursive solution calls itself
- Each recursive call solves an identical, smaller problem
- Test for base case enables recursive calls to stop
- Eventually one of smaller calls will be base case

A Recursive Valued Function

The factorial of *n*

```
/** Computes the factorial of the nonnegative integer n.
 @pre  n must be greater than or equal to 0.
 @post  None.
 @return  The factorial of n; n is unchanged. */
int fact(int n)
{
   if (n == 0)
      return 1;
   else // n > 0, so n-1 >= 0. Thus, fact(n-1) returns (n-1)!
      return n * fact(n - 1); // n * (n-1)! is n!
}  //  end fact
```

A Recursive Valued Function

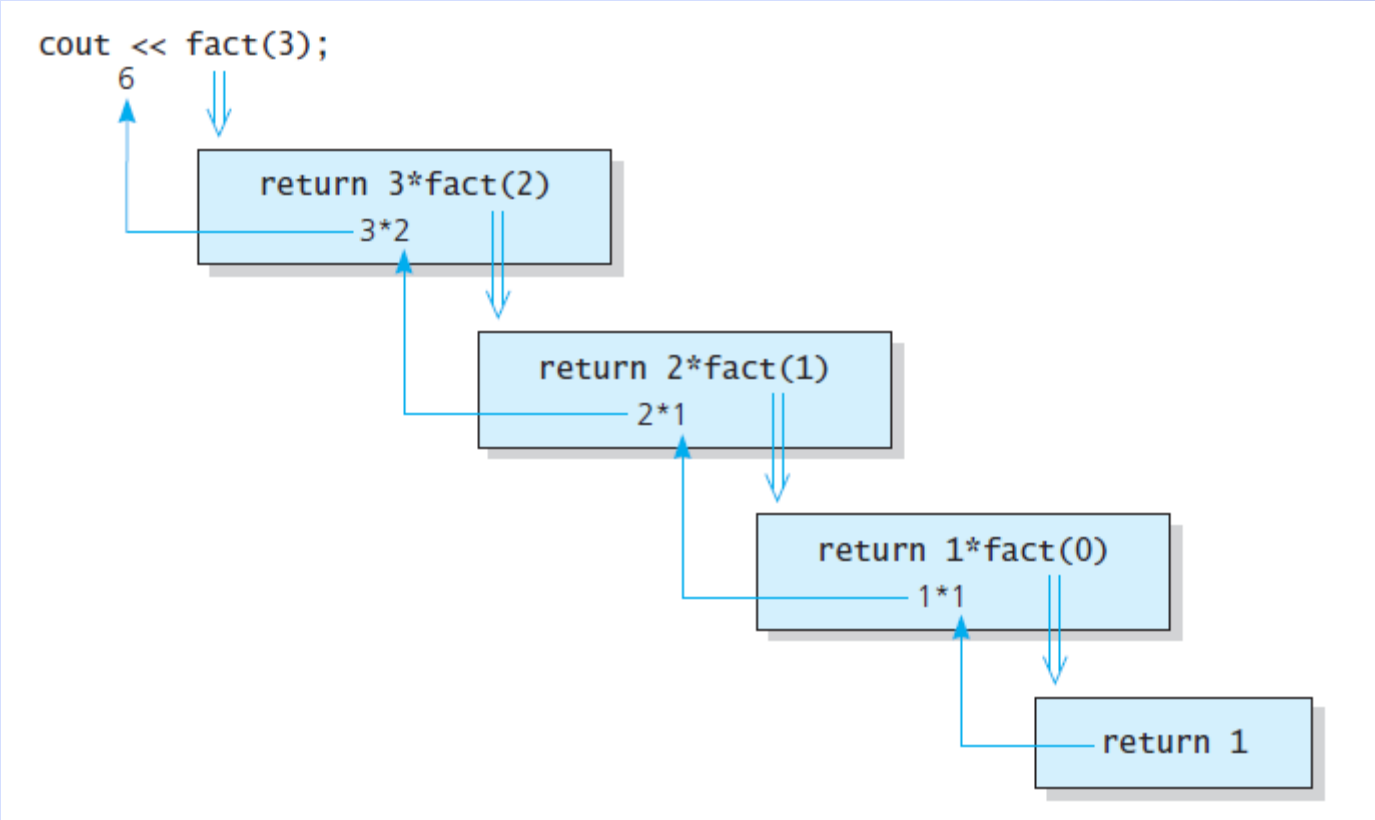


FIGURE 2-2 **`fact(3)`**

The Box Trace

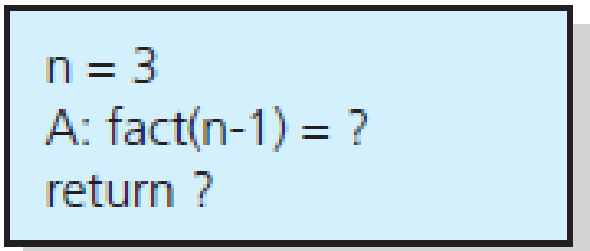


FIGURE 2-3 A box

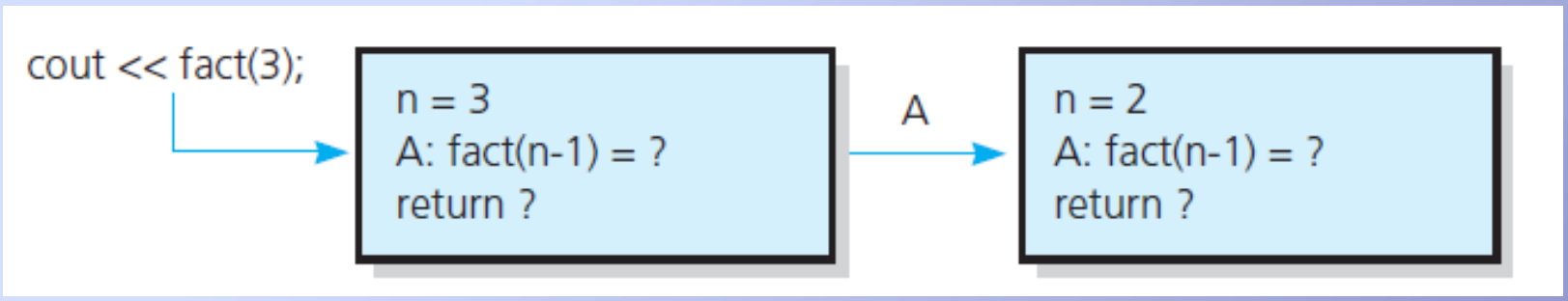
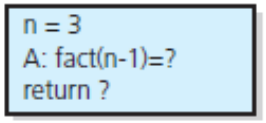


FIGURE 2-4 The beginning of the box trace

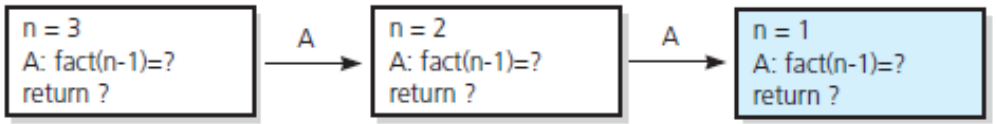
The Box Trace

The initial call is made, and method `fact` begins execution:



At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

At point A a recursive call is made, and the new invocation of the method `fact` begins execution:



At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

(continues)

FIGURE 2-5 Box trace of fact(3)

The Box Trace

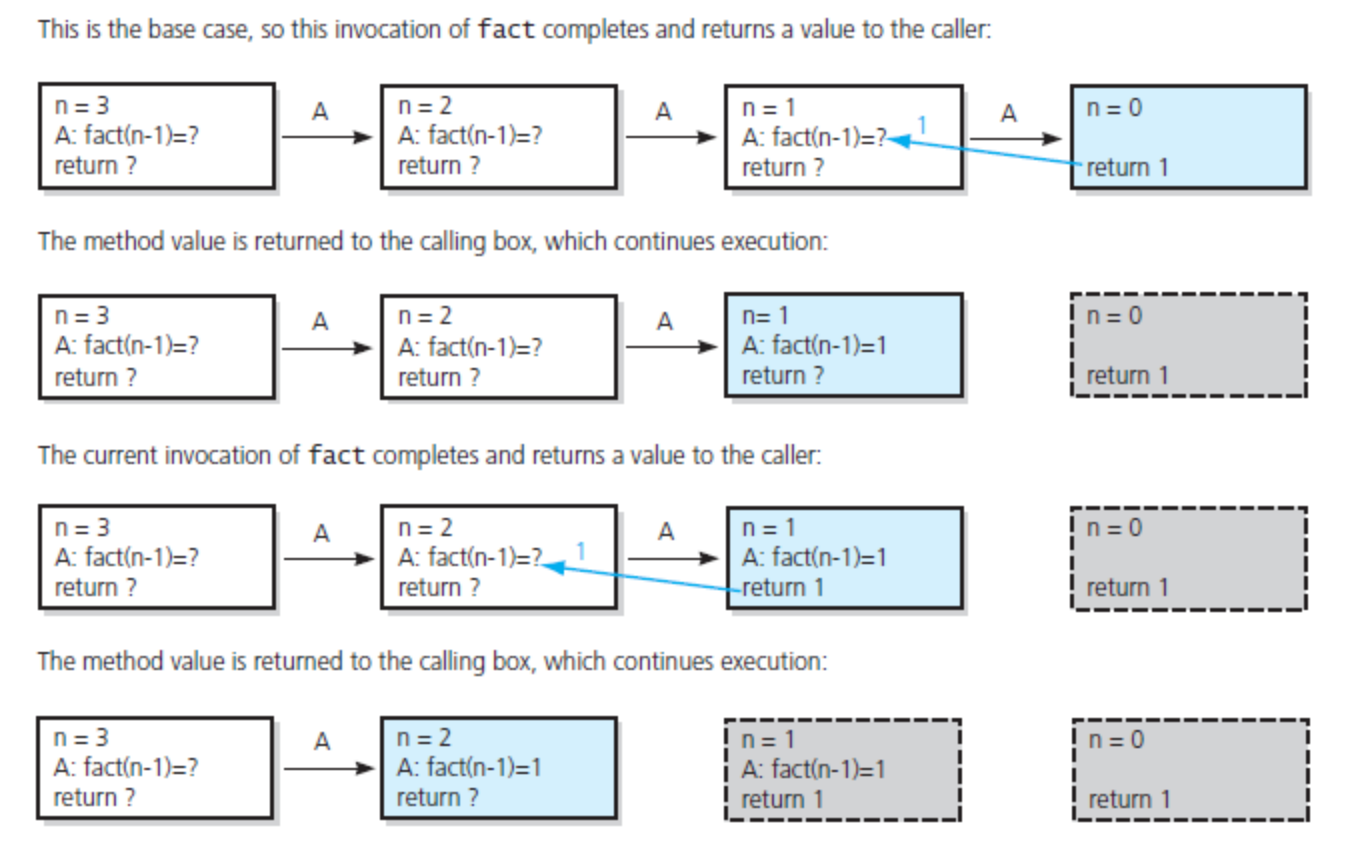


FIGURE 2-5 Box trace of fact(3) … continued

The Box Trace

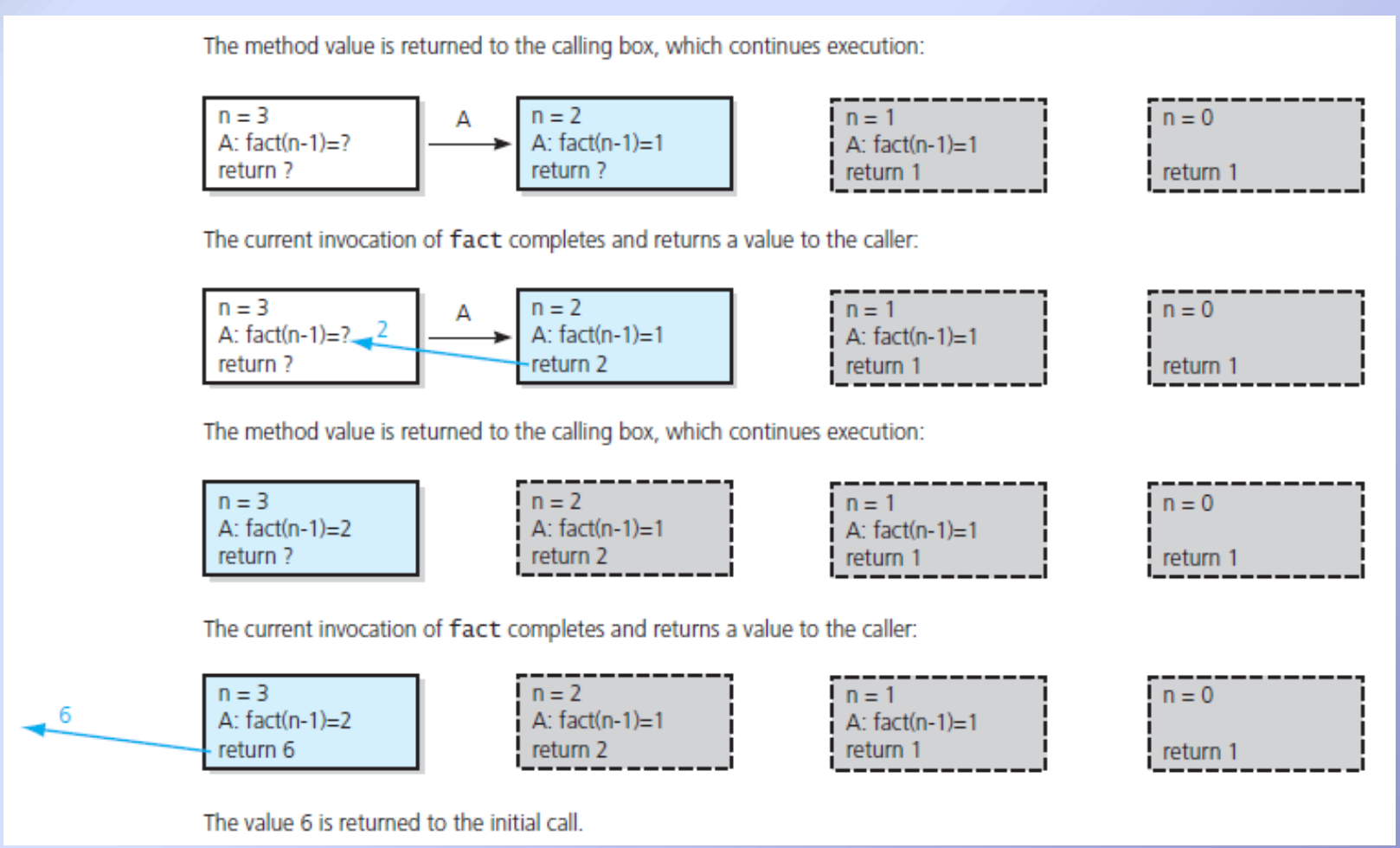


FIGURE 2-5 Box trace of fact(3) … continued

A Recursive Void Function

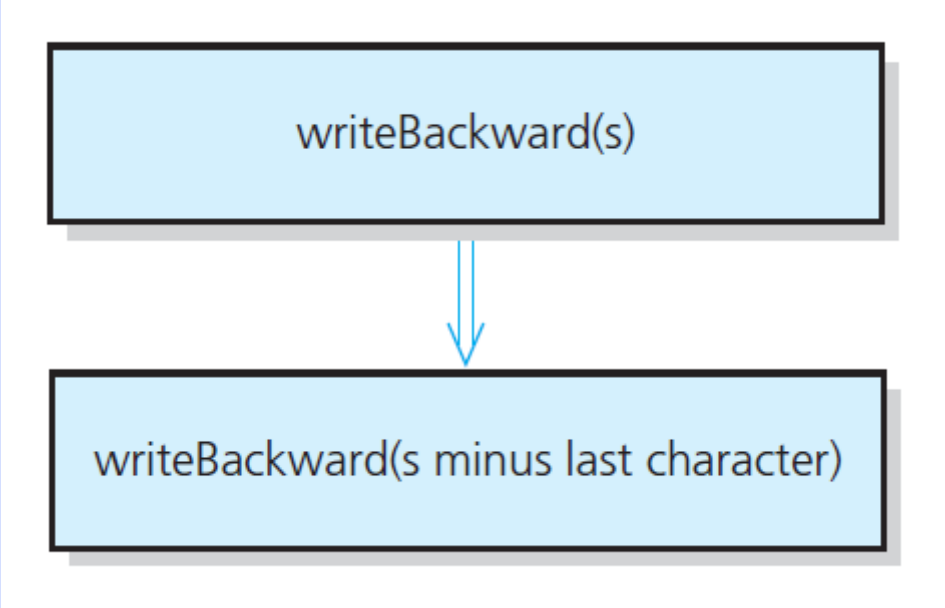


FIGURE 2-6 A recursive solution

A Recursive Void Function

- The function **`writeBackwards`**

```
/** Writes a character string backward.
 @pre  The string s to write backward.
 @post  None.
   @param s  The string to write backward. */
 void writeBackward(string s)
 {
    int length = s.size(); // Length of string
    if (length > 0)
    {
       // Write the last character
       cout << s.substr(length - 1, 1);

       // Write the rest of the string backward
       writeBackward(s.substr(0, length - 1)); // Point A
    }  // end if

    // length == 0 is the base case - do nothing
 }  // end writeBackward
```

A Recursive Void Function

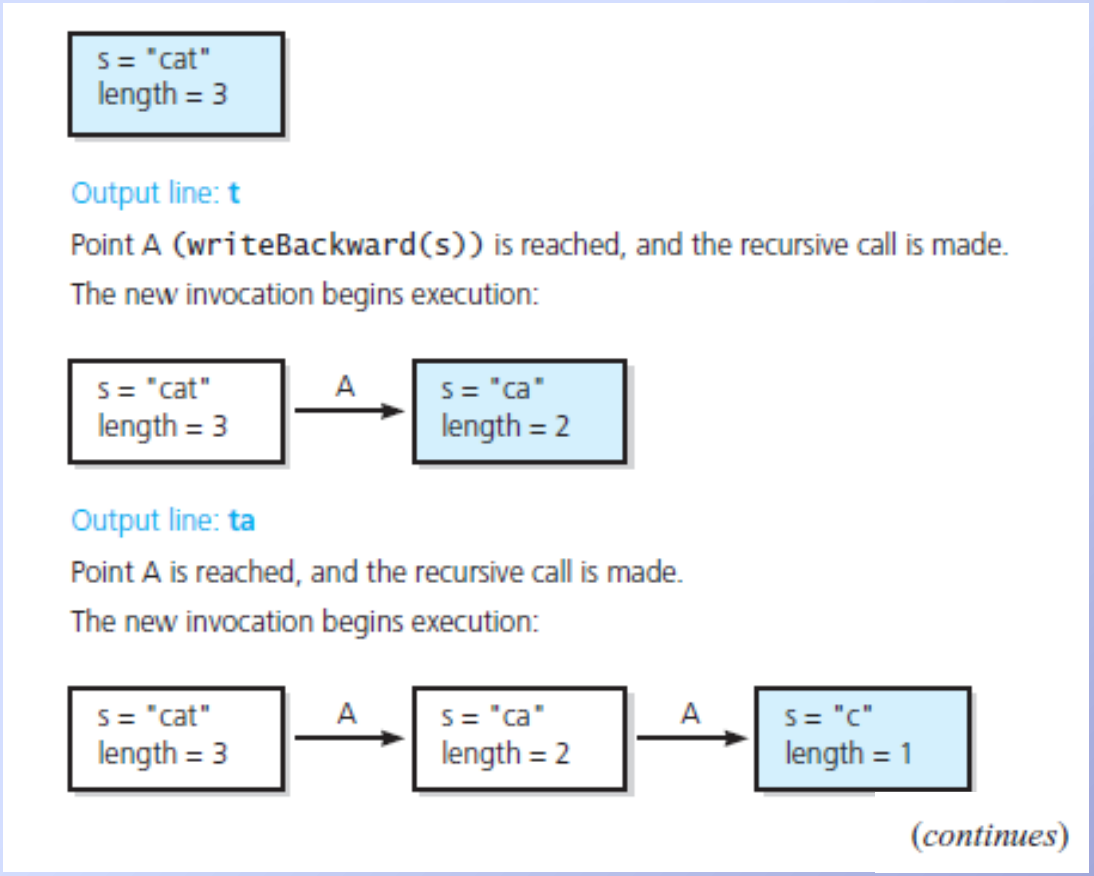


FIGURE 2-7 Box trace of **`writeBackward("cat")`**

A Recursive Void Function

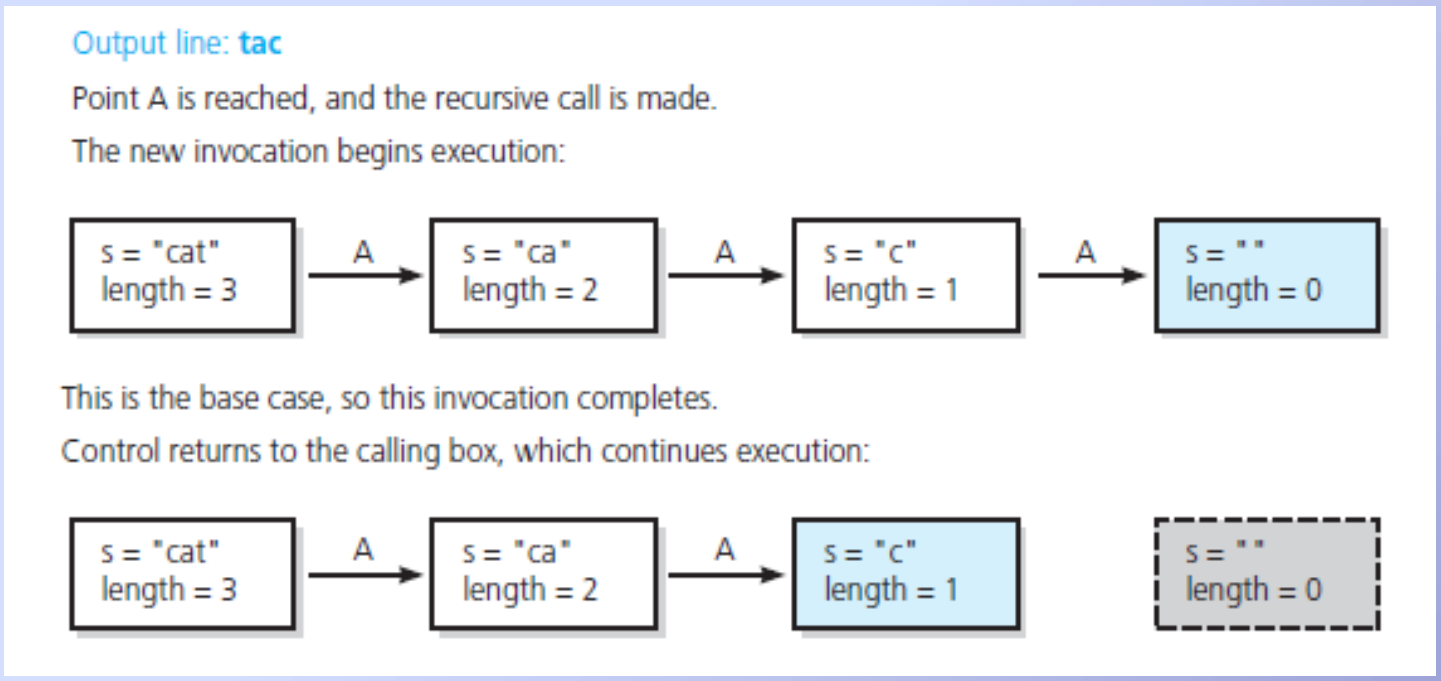


FIGURE 2-7 **`writeBackward("cat")`** continued

A Recursive Void Function

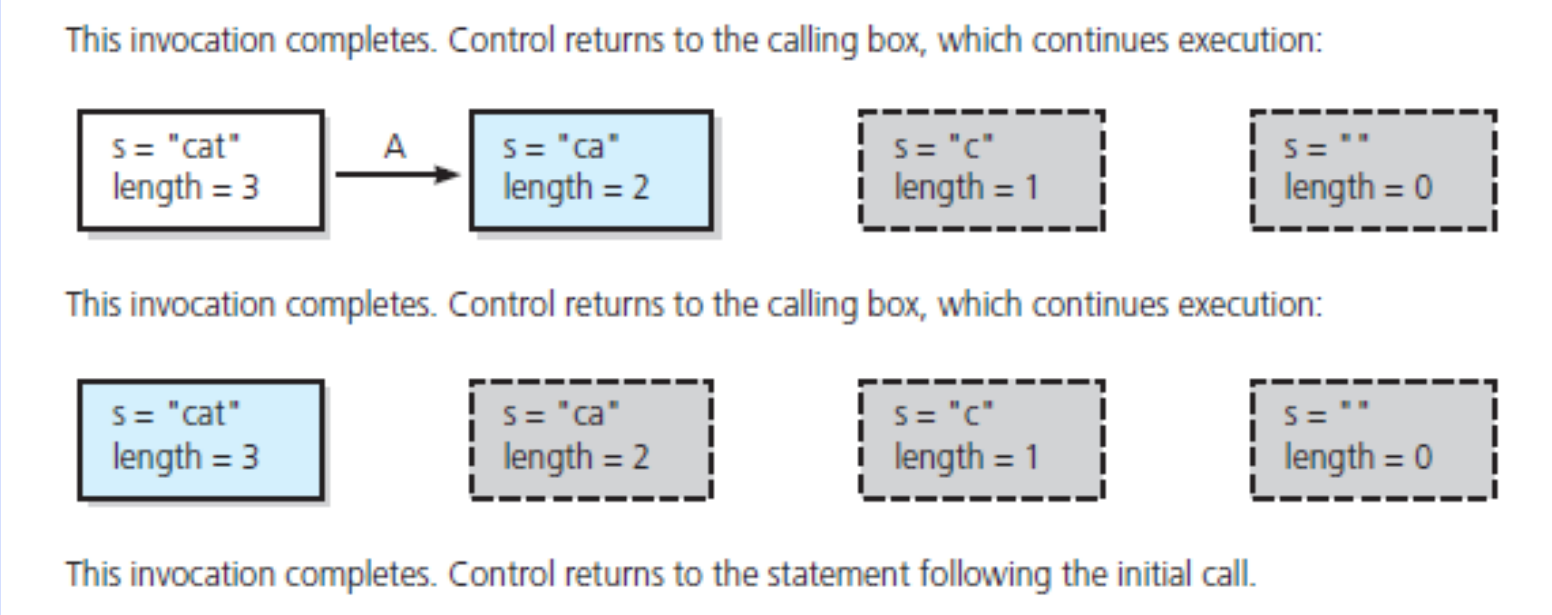
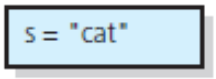


FIGURE 2-7 **`writeBackward("cat")`** continued

A Recursive Void Function

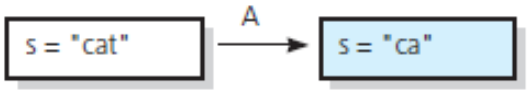
The initial call is made, and the function begins execution:



Output stream:

```
Enter writeBackward with string: cat
About to write last character of string: cat
t
```

Point A is reached, and the recursive call is made. The new invocation begins execution:



Output stream:

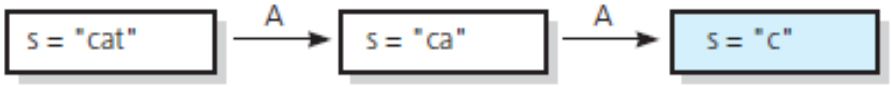
```
Enter writeBackward with string: cat
About to write last character of string: cat
t
Enter writeBackward with string: ca
About to write last character of string: ca
a
```

(continues)

FIGURE 2-8 Box trace of **`writeBackward("cat")`** in pseudocode

A Recursive Void Function

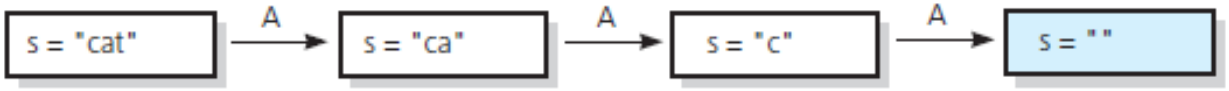
Point A is reached, and the recursive call is made. The new invocation begins execution:



Output stream:

```
Enter writeBackward with string: cat
About to write last character of string: cat
t
Enter writeBackward with string: ca
About to write last character of string: ca
a
Enter writeBackward with string: c
About to write last character of string: c
c
```

Point A is reached, and the recursive call is made. The new invocation begins execution:



This invocation completes execution, and a return is made.

(continues)

FIGURE 2-8 Box trace of **`writeBackward("cat")`** in pseudocode (continued)

A Recursive Void Function

Output stream:

```
Enter writeBackward with string: cat
About to write last character of string: cat
t
Enter writeBackward with string: ca
About to write last character of string: ca
a
Enter writeBackward with string: c
About to write last character of string: c
c
Enter writeBackward with string:
Leave writeBackward with string:
```

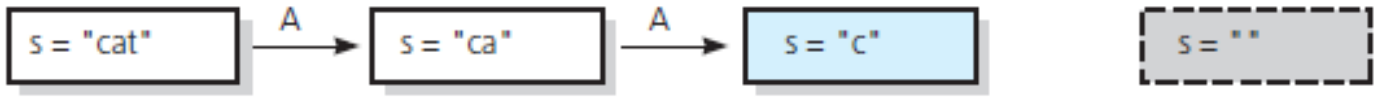


FIGURE 2-8 Box trace of **`writeBackward("cat")`** in pseudocode (continued)

A Recursive Void Function

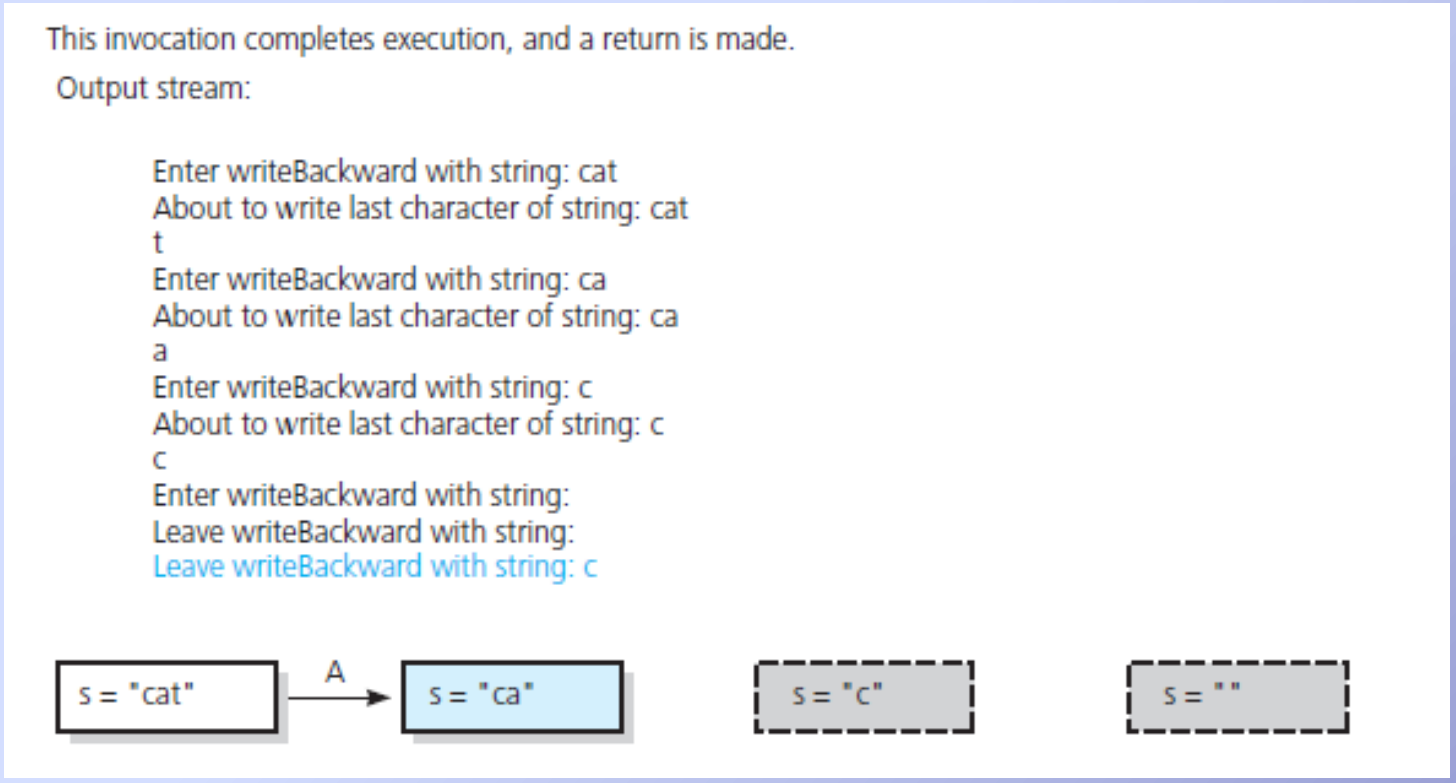


FIGURE 2-8 Box trace of **`writeBackward("cat")`** in pseudocode (continued)

A Recursive Void Function

This invocation completes execution, and a return is made.

Output stream:

```
Enter writeBackward with string: cat
About to write last character of string: cat
t
Enter writeBackward with string: ca
About to write last character of string: ca
a
Enter writeBackward with string: c
About to write last character of string: c
```

(continues)

FIGURE 2-8 Box trace of **`writeBackward("cat")`** in pseudocode (continued)

A Recursive Void Function

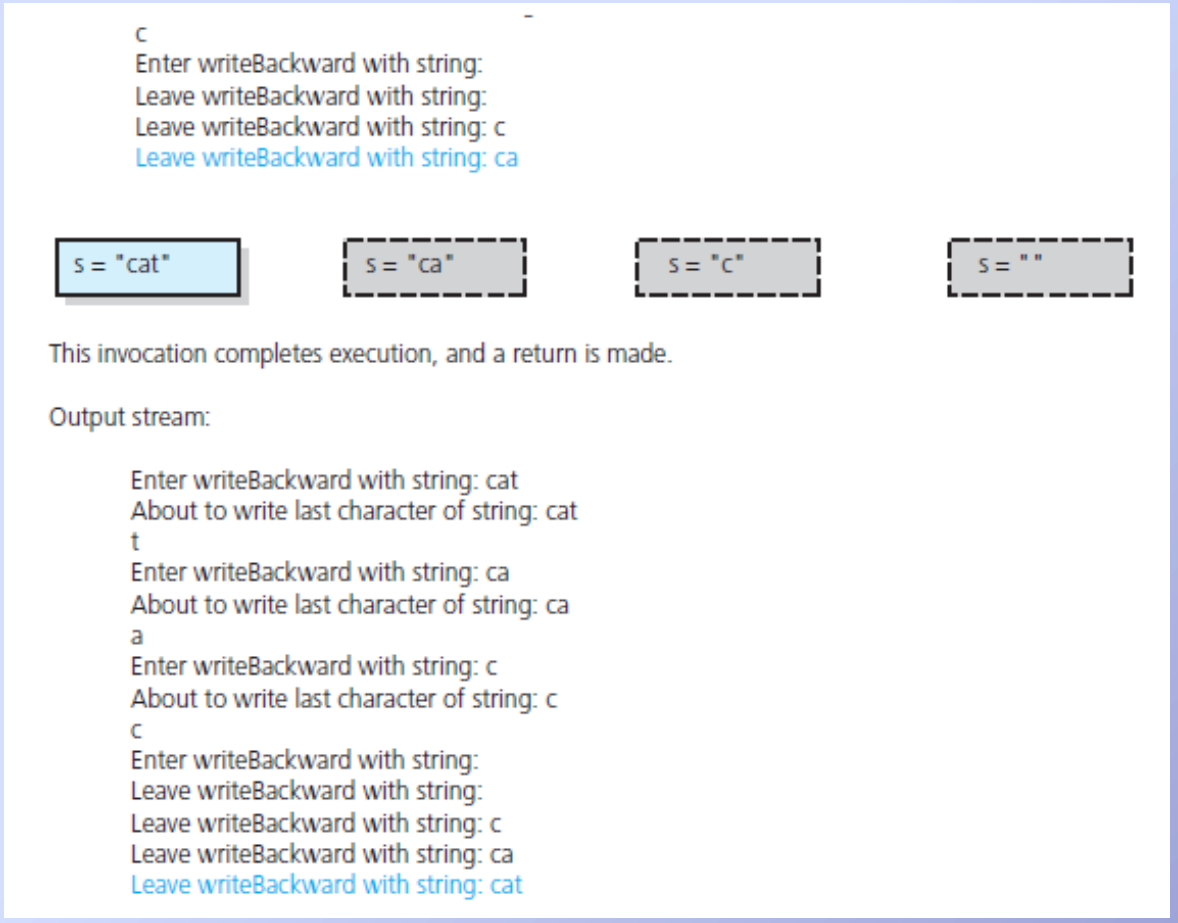


FIGURE 2-8 Box trace of **`writeBackward("cat")`** in pseudocode (concluded)

A Recursive Void Function

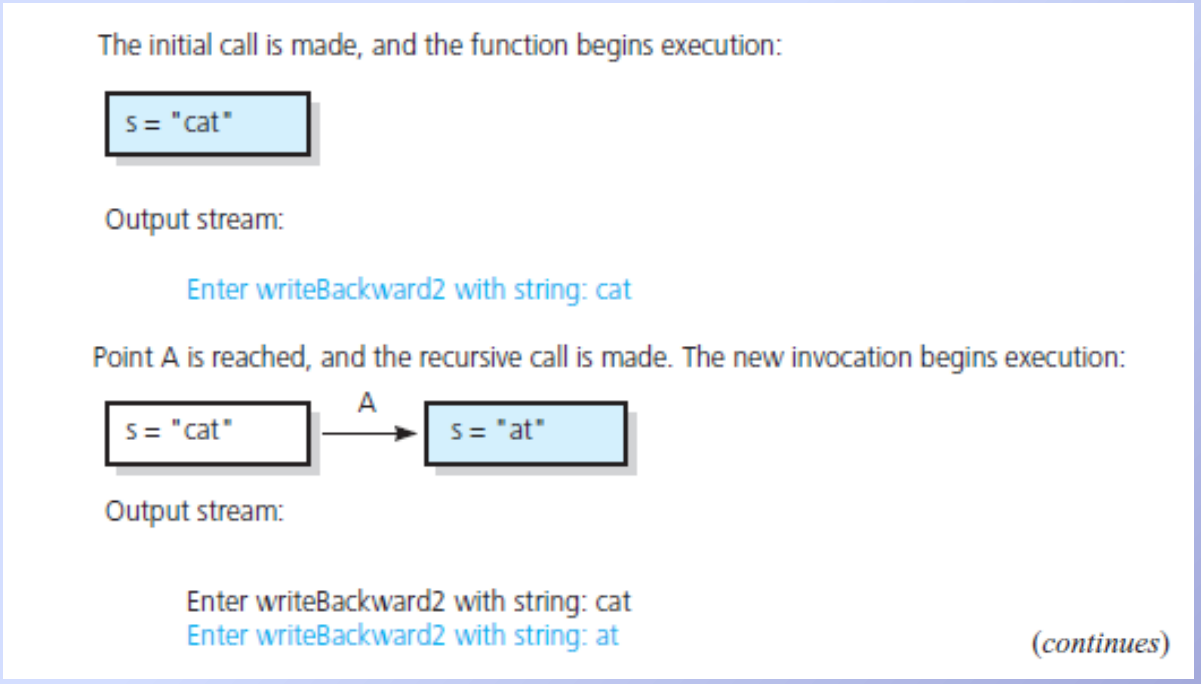
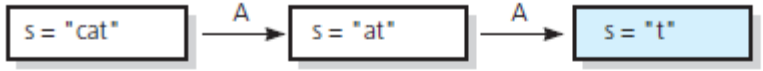


FIGURE 2-9 Box trace of **`writeBackward2("cat")`** in pseudocode

A Recursive Void Function

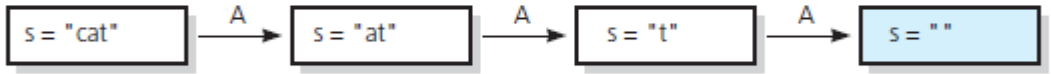
Point A is reached, and the recursive call is made. The new invocation begins execution:



Output stream:

```
Enter writeBackward2 with string: cat
Enter writeBackward2 with string: at
Enter writeBackward2 with string: t
```

Point A is reached, and the recursive call is made. The new invocation begins execution:



This invocation completes execution, and a return is made.

Output stream:

```
Enter writeBackward2 with string: cat
Enter writeBackward2 with string: at
Enter writeBackward2 with string: t
Enter writeBackward2 with string:
Leave writeBackward2 with string:
```

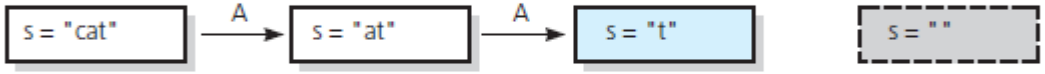
(continues)

FIGURE 2-9 Box trace of **`writeBackward2("cat")`** in pseudocode (continued)

A Recursive Void Function

Output stream:

Enter writeBackward2 with string: cat
Enter writeBackward2 with string: at
Enter writeBackward2 with string: t
Enter writeBackward2 with string:
Leave writeBackward2 with string:



This invocation completes execution, and a return is made.

Output stream:

Enter writeBackward2 with string: cat
Enter writeBackward2 with string: at
Enter writeBackward2 with string: t

(continues)

FIGURE 2-9 Box trace of **`writeBackward2("cat")`** in pseudocode (continued)

A Recursive Void Function

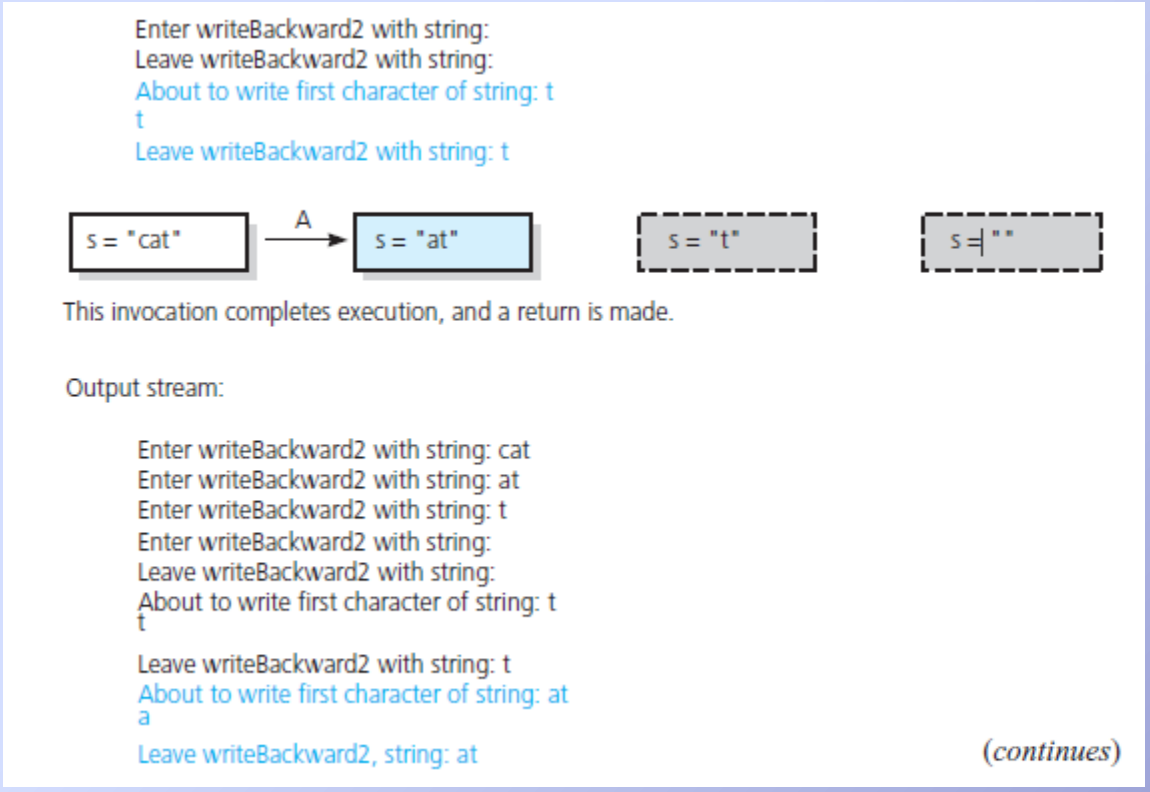


FIGURE 2-9 Box trace of **`writeBackward2("cat")`** in pseudocode (continued)

A Recursive Void Function

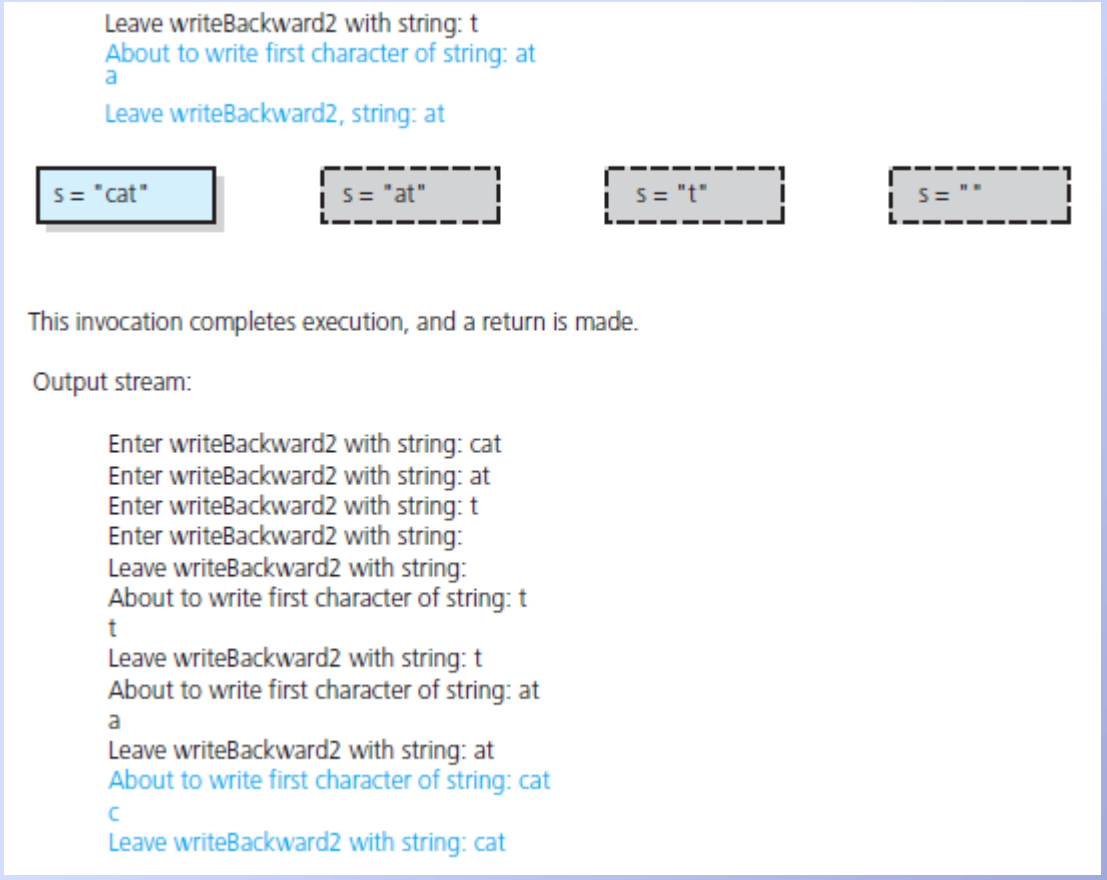


FIGURE 2-9 Box trace of **`writeBackward2("cat")`** in pseudocode (concluded)

Writing an Array's Entries in Backward Order

- Pseudocode

```
writeArrayBackward(anArray: char[])

    if (the array is empty)
        Do nothing—this is the base case
    else
    {
        Write the last character in anArray
        writeArrayBackward(anArray minus its last character)
    }
```

Writing an Array's Entries in Backward Order

- Source code

```
/** Writes the characters in an array backward.
  @pre  The array anArray contains size characters, where size >= 0.
  @post  None.
  @param anArray  The array to write backward.
  @param first  The index of the first character in the array.
  @param last  The index of the last character in the array. */
void writeArrayBackward(const char anArray[], int first, int last)
{
   if (first <= last)
   {
      // Write the last character
      cout << anArray[last];

      // Write the rest of the array backward
      writeArrayBackward(anArray, first, last - 1);
   }  // end if

   // first > last is the base case - do nothing
}  // end writeArrayBackward
```

The Binary Search

- A high-level binary search for the array problem

```
binarySearch(anArray: ArrayType, target: ValueType)

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

The Binary Search

Issues to consider

1. How to pass a half array to recursive call
2. How to determine which half of array has target value
3. What is the base case?
4. How will result of binary search be indicated?

The Binary Search

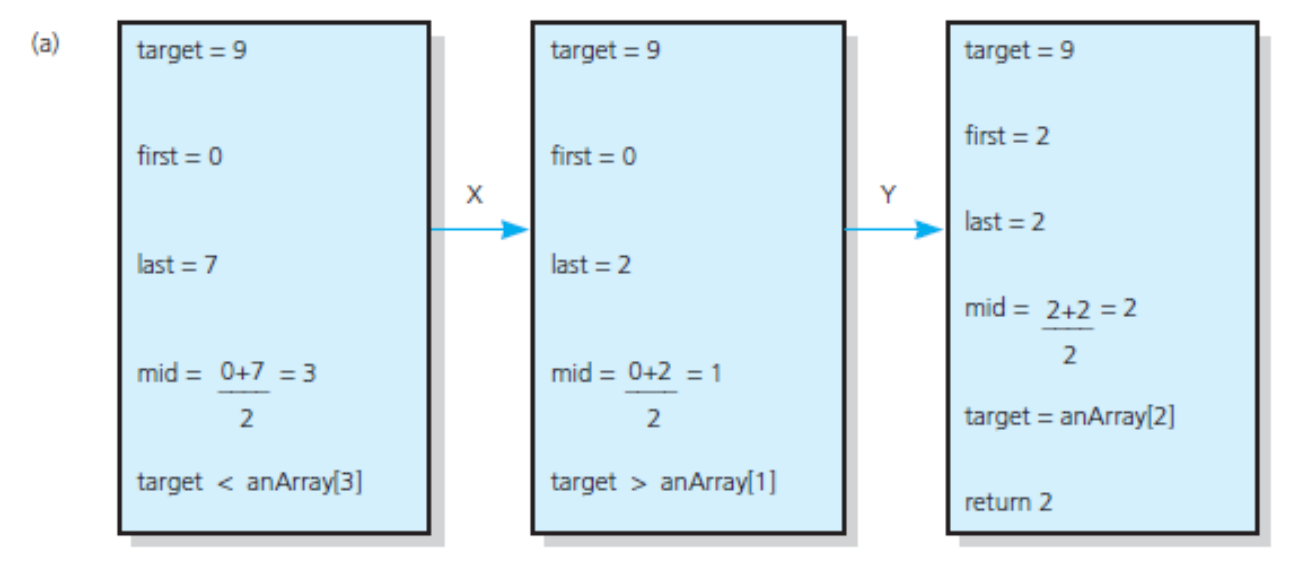


FIGURE 2-10 Box traces of **`binarySearch`** with **`anArray`** = <1, 5, 9, 12, 15, 21, 29, 31>:
(a) a successful search for 9

The Binary Search

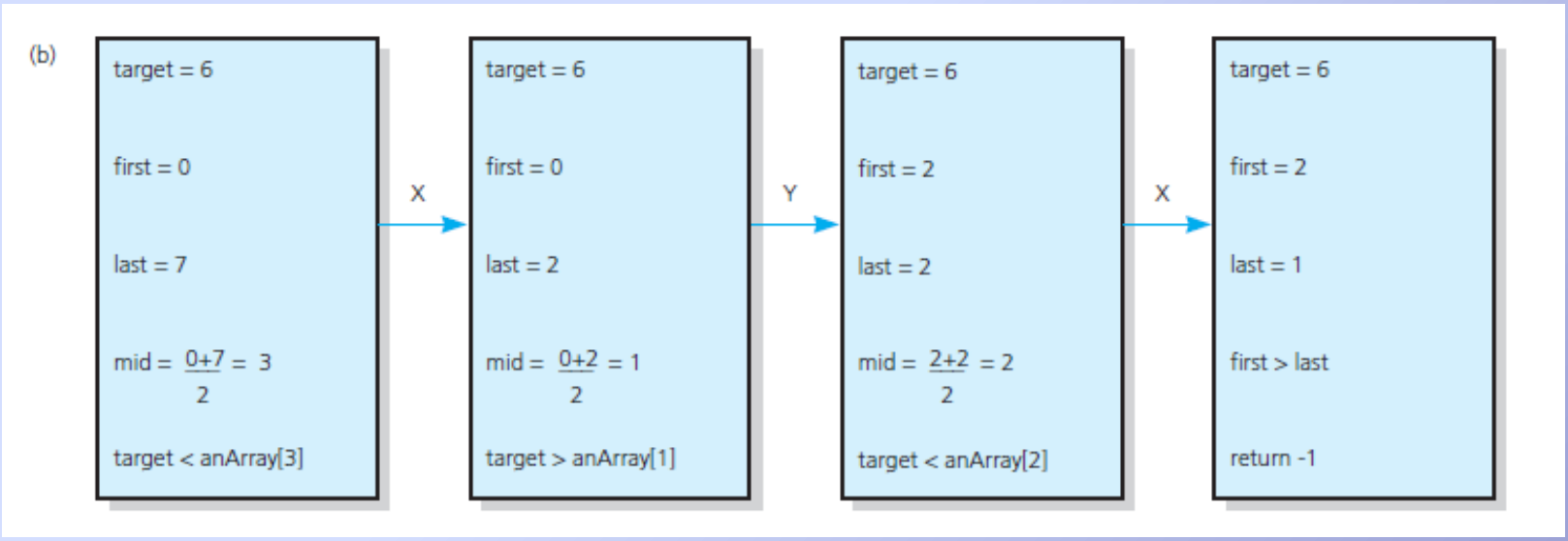


FIGURE 2-10 Box traces of **`binarySearch`** with **`anArray`** = <1, 5, 9, 12, 15, 21, 29, 31>:
(b) an unsuccessful search for 6

The Binary Search

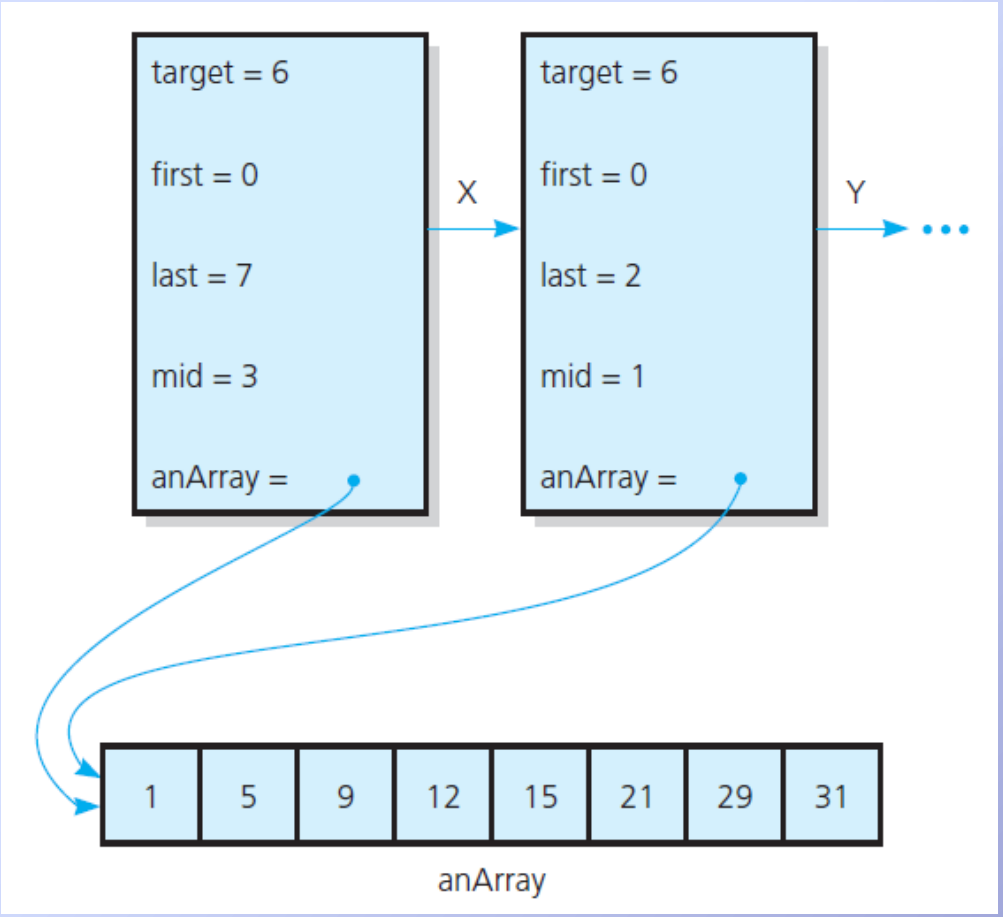


FIGURE 2-11 Box trace with a reference argument

Finding the Largest Value in an Array

- Recursive algorithm

```
if (anArray has only one entry)
    maxArray(anArray) is the entry in anArray
else if (anArray has more than one entry)
    maxArray(anArray) is the maximum of
        maxArray(left half of anArray) and maxArray(right half of anArray)
```

Finding the Largest Value in an Array

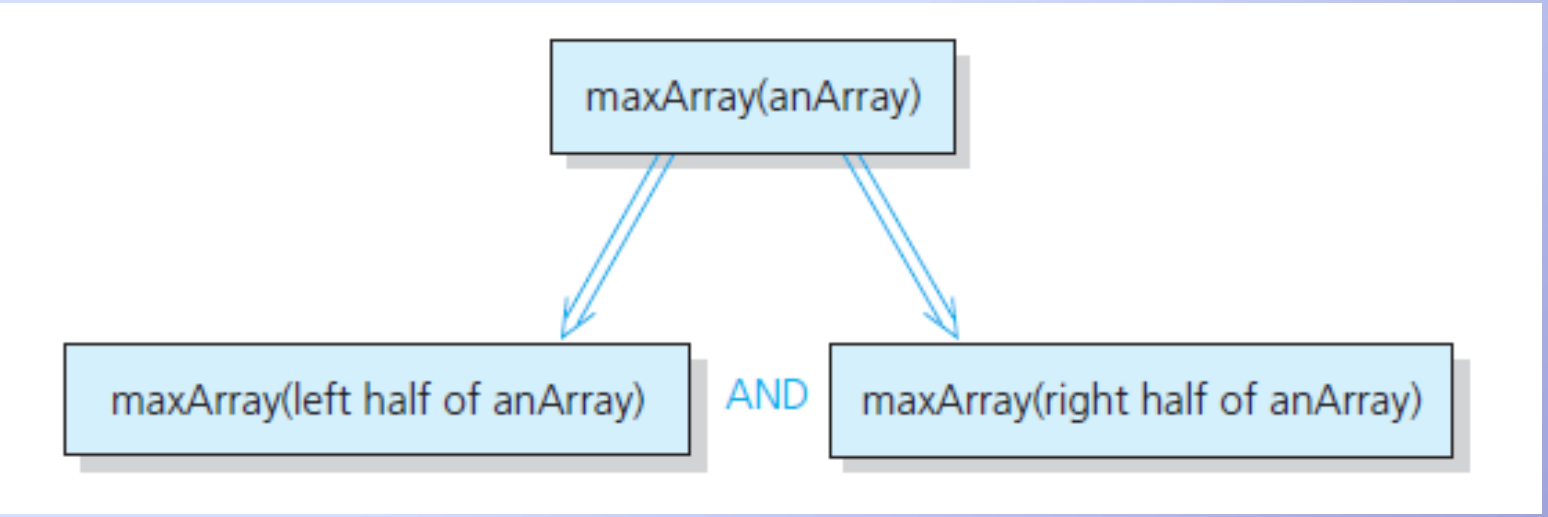


FIGURE 2-12 Recursive solution to the largest-value problem

Finding the Largest Value in an Array

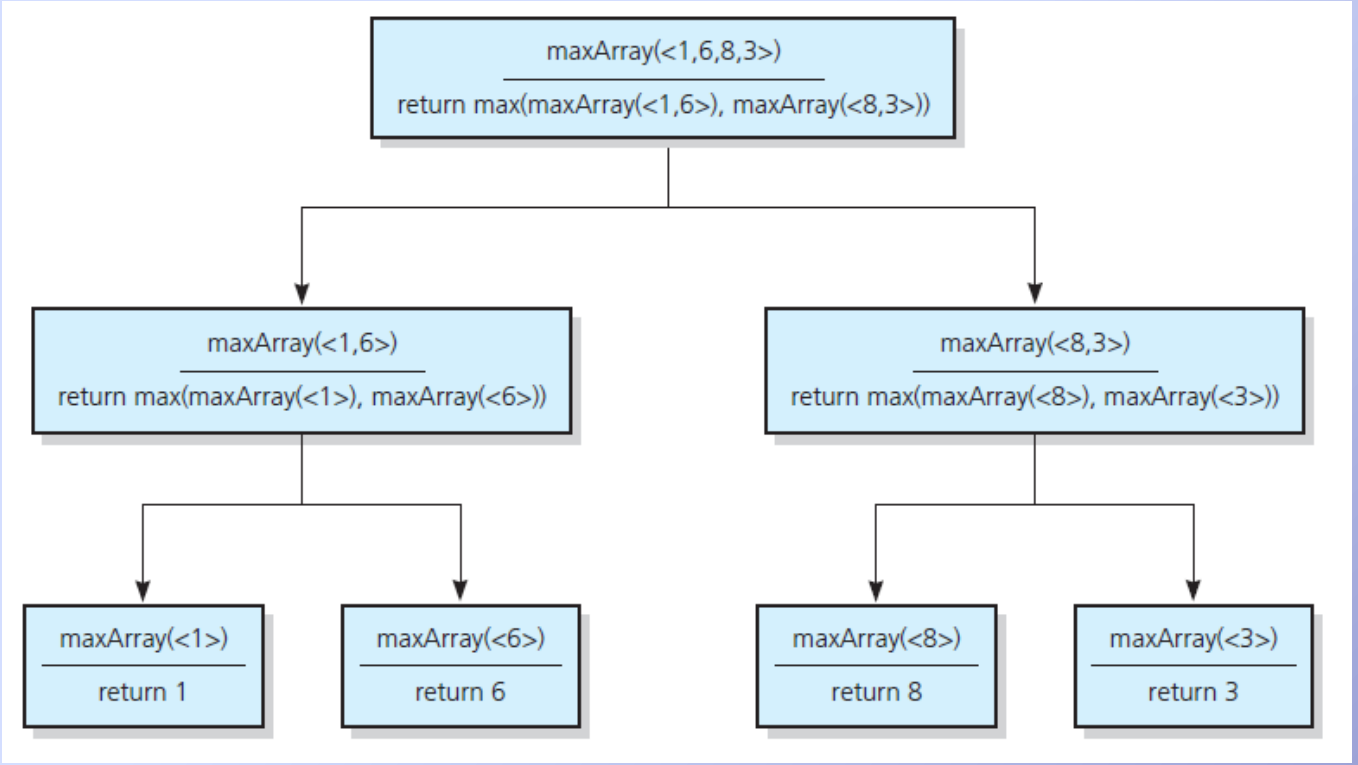


FIGURE 2-13 The recursive calls that **`maxArray(<1,6,8,3>)`** generates

Finding the k^{th} Smallest Value of an Array

The recursive solution proceeds by:

1. Selecting a pivot value in array
2. Cleverly arranging/partitioning, values in array about this pivot value
3. Recursively applying strategy to one of partitions

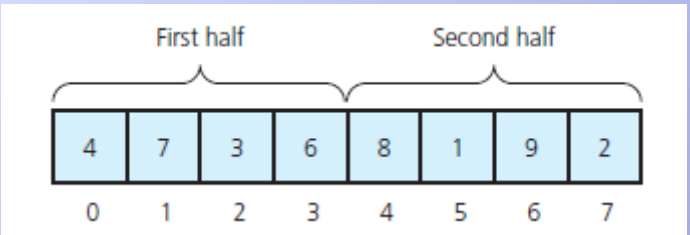


FIGURE 2-14 A sample array

Finding the k^{th} Smallest Value of an Array

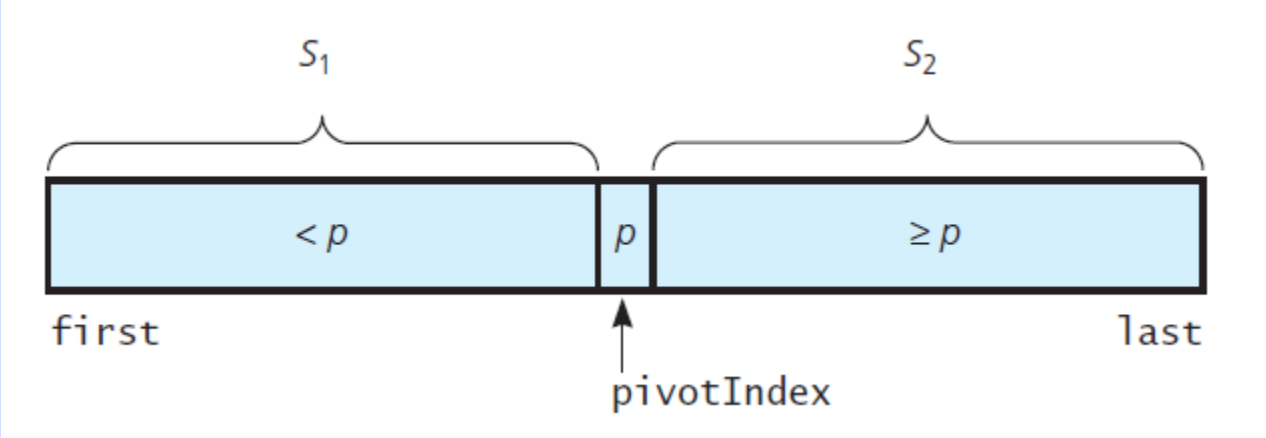


FIGURE 2-15 A partition about a pivot

Finding the k^{th} Smallest Value of an Array

High level pseudocode solution:

```
// Returns the kth smallest value in anArray[first..last].

kSmall(k: integer, anArray: ArrayType,
       first: integer, last: integer): ValueType

   Choose a pivot value p from anArray[first..last]
   Partition the values of anArray[first..last] about p

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

Organizing Data
Towers of Hanoi

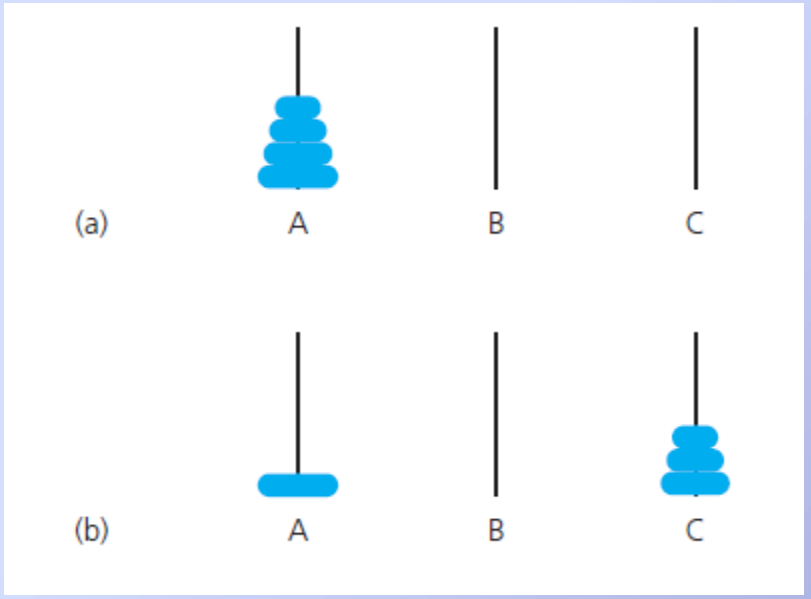


FIGURE 2-16 (a) The initial state;
(b) move n – 1 disks from A to C;

Towers of Hanoi

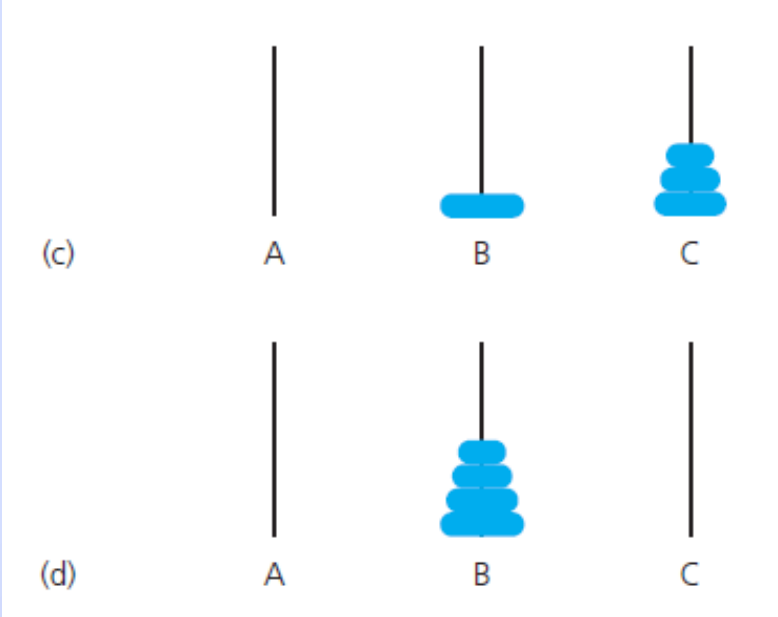


FIGURE 2-16 (c) move 1 disk from A to B;
(d) move n – 1 disks from C to B

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)
   }
```

Towers of Hanoi

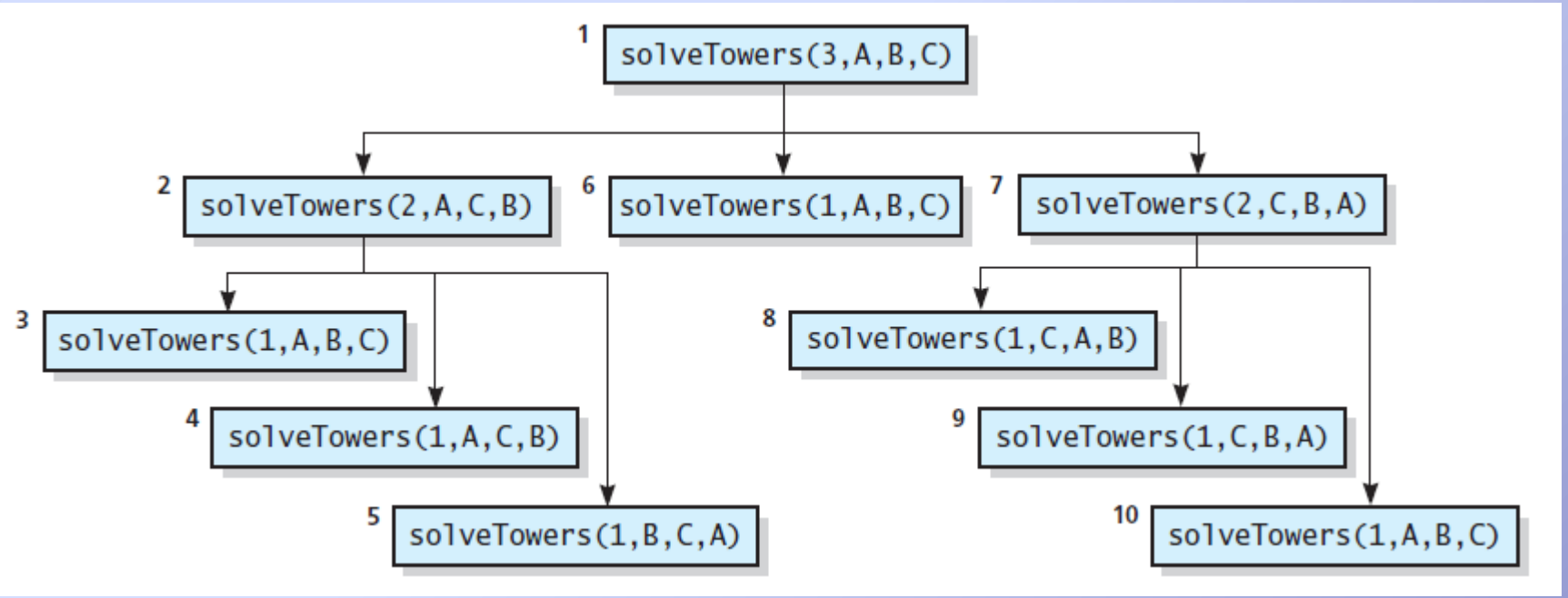


FIGURE 2-17 The order of recursive calls that results from solve **`Towers(3, A, B, C)`**

Towers of Hanoi

- Source code for **`solveTowers`**

```
void solveTowers(int count, char source, char destination, char spare)
{
   if (count == 1)
   {
      cout << "Move top disk from pole " << source
           << " to pole " << destination << endl;
   }
   else
   {
      solveTowers(count - 1, source, spare, destination);  // X
      solveTowers(1, source, destination, spare);          // Y
      solveTowers(count - 1, spare, destination, source);  // Z
   }  // end if
}  // end solveTowers
```

The Fibonacci Sequence (Multiplying Rabbits)

Assumed “facts” about rabbits:

- Rabbits never die.
- A rabbit reaches sexual maturity exactly two months after birth
- Rabbits 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.

The Fibonacci Sequence (Multiplying Rabbits)

Month	Rabbit Population
1	One pair
2	One pair, still
3	Two pairs
4	Three pairs
5	Five pairs
6	8 pairs

The Fibonacci Sequence (Multiplying Rabbits)

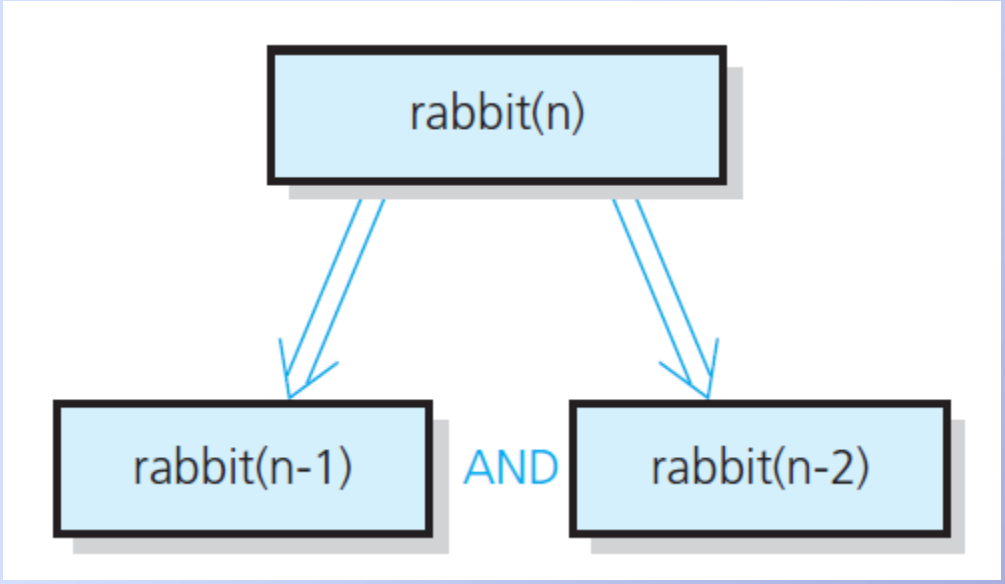


FIGURE 2-18 Recursive solution to the rabbit problem

The Fibonacci Sequence (Multiplying Rabbits)

- A C++ function to compute **`rabbit(n)`**

```
/** Computes a term in the Fibonacci sequence.
  @pre  n is a positive integer.
  @post  None.
  @param n  The given integer.
  @return  The nth Fibonacci number. */
int rabbit(int n)
{
   if (n <= 2)
      return 1;
   else // n > 2, so n - 1 > 0 and n - 2 > 0
      return rabbit(n - 1) + rabbit(n - 2);
}  // end rabbit
```

The Fibonacci Sequence (Multiplying Rabbits)

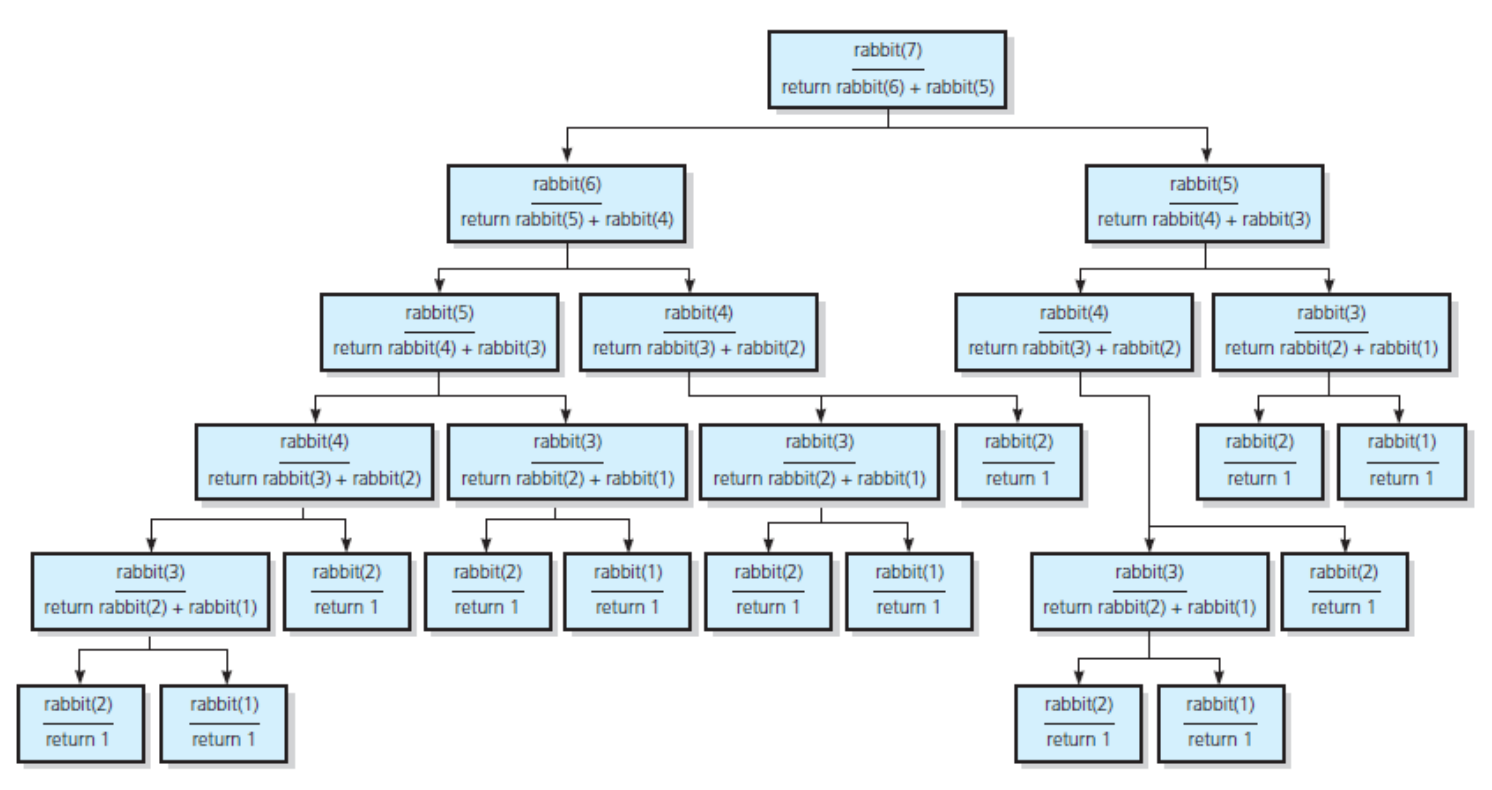


FIGURE 2-19 The recursive calls that rabbit(7) generates

Choosing *k* Out of *n* Things

- Recursive solution:

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

Choosing *k* Out of *n* Things

- Recursive function:

```
/** Computes the number of groups of k out of n things.
 @pre  n and k are nonnegative integers.
 @post  None.
 @param n  The given number of things.
 @param k  The given number to choose.
 @return  g(n, k). */
int getNumberOfGroups(int n, int k)
{
   if ( (k == 0) || (k == n) )
      return 1;
   else if (k > n)
      return > 0;
   else
      return g(n - 1, k - 1) + g(n - 1, k);
}  // end getNumberOfGroups
```

Choosing *k* Out of *n* Things

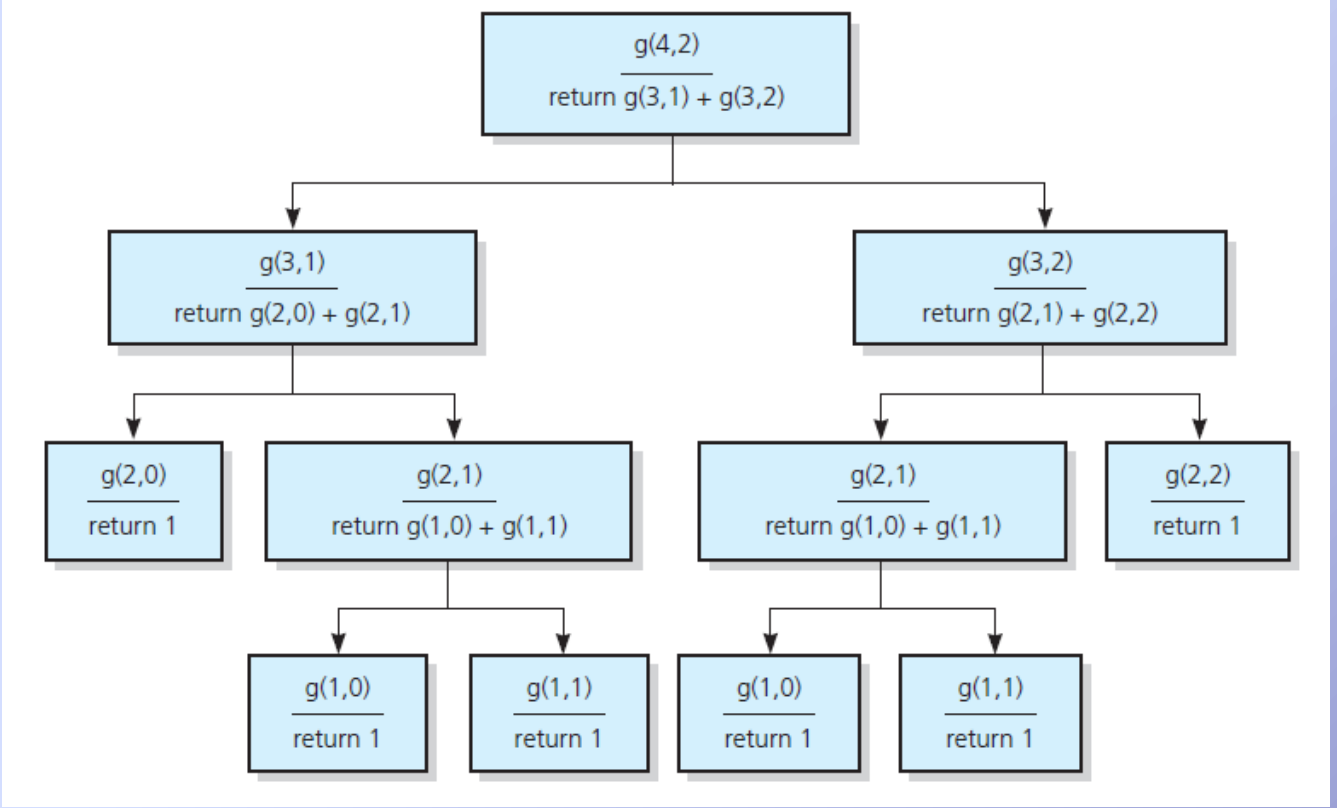
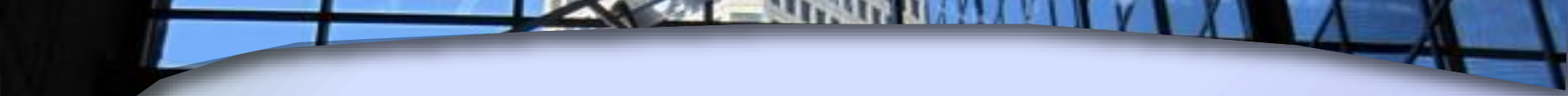


FIGURE 2-20 The recursive calls that g (4, 2) generates

Recursion and Efficiency

- Inefficiency factors
  - Overhead associated with function calls
  - Inherent inefficiency of some recursive algorithms
- Principle:
  - Do not use recursive solution if inefficient and clear, efficient iterative solution exists



End

Chapter 2