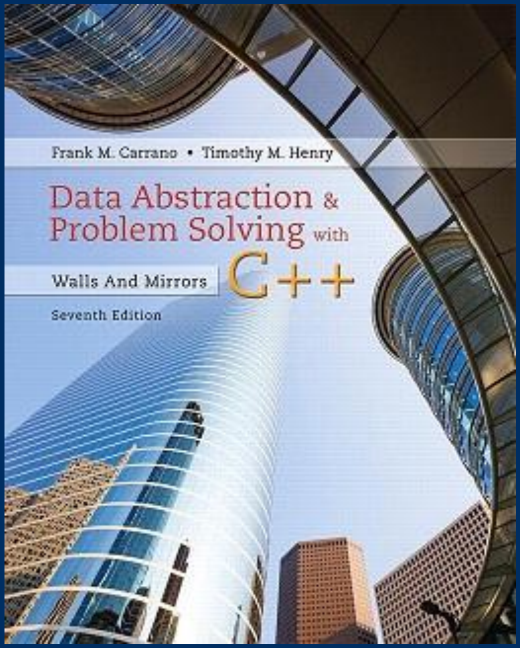
Chapter 2

Recursion: The Mirrors

CS 302 - Data Structures

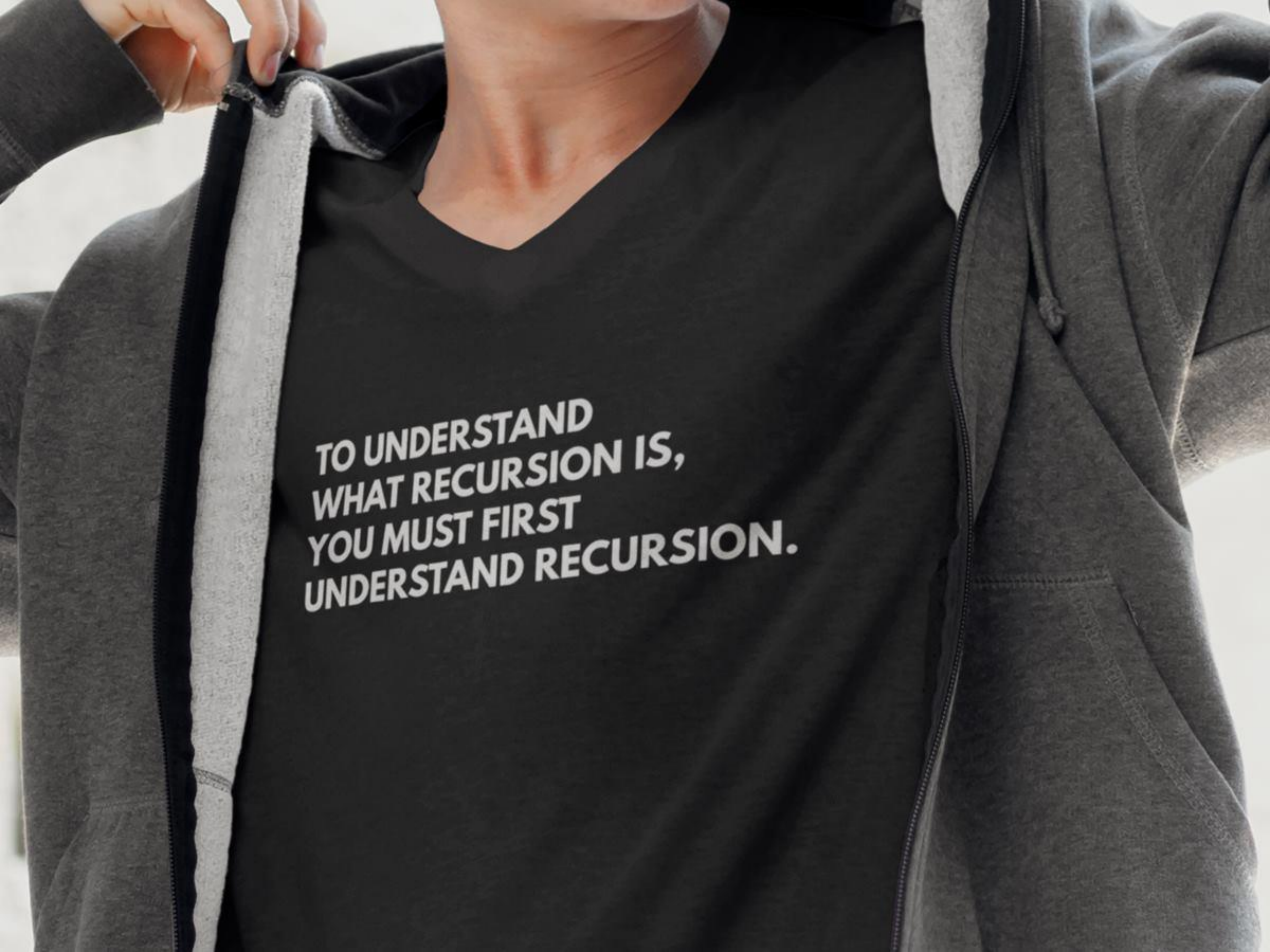
M. Abdullah Canbaz



Modified from authors’ slides

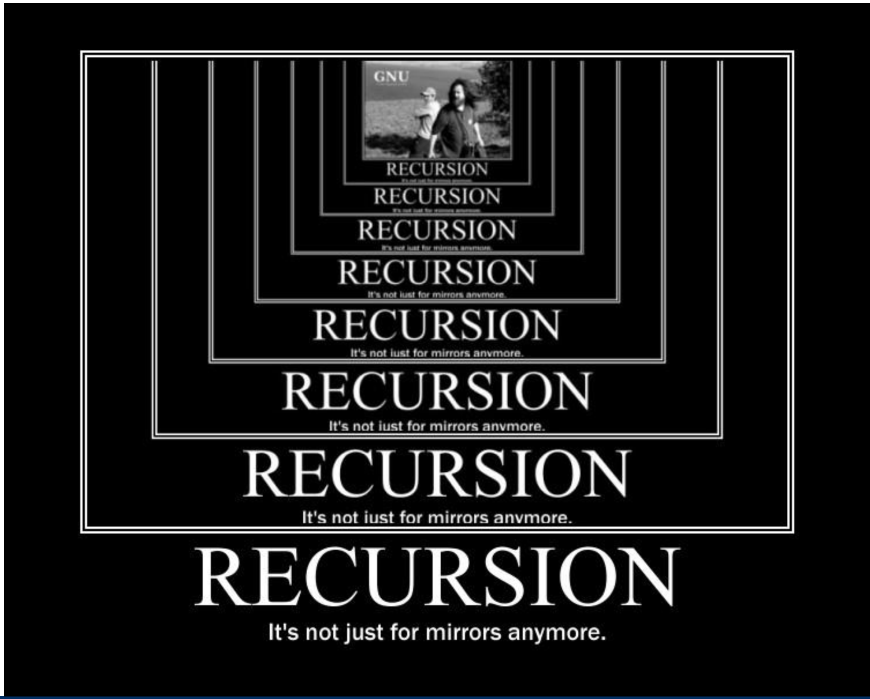
Reminders

- Assignment 1 is due
  - Monday February 5th at 2pm.
  - Deliverables:
    - A doc or PDF (consists of the CRC card and UML Class Diagram)
    - Doxygen Documentation
  - **TA**: Athanasia Katsila,
    Email: *akatsila [at] nevada {dot} unr {dot} edu,*
    Office Hours: Thursdays, 10:30 am - 12:30 pm at SEM 211


TO UNDERSTAND
WHAT RECURSION IS,
YOU MUST FIRST
UNDERSTAND RECURSION.

Contents

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



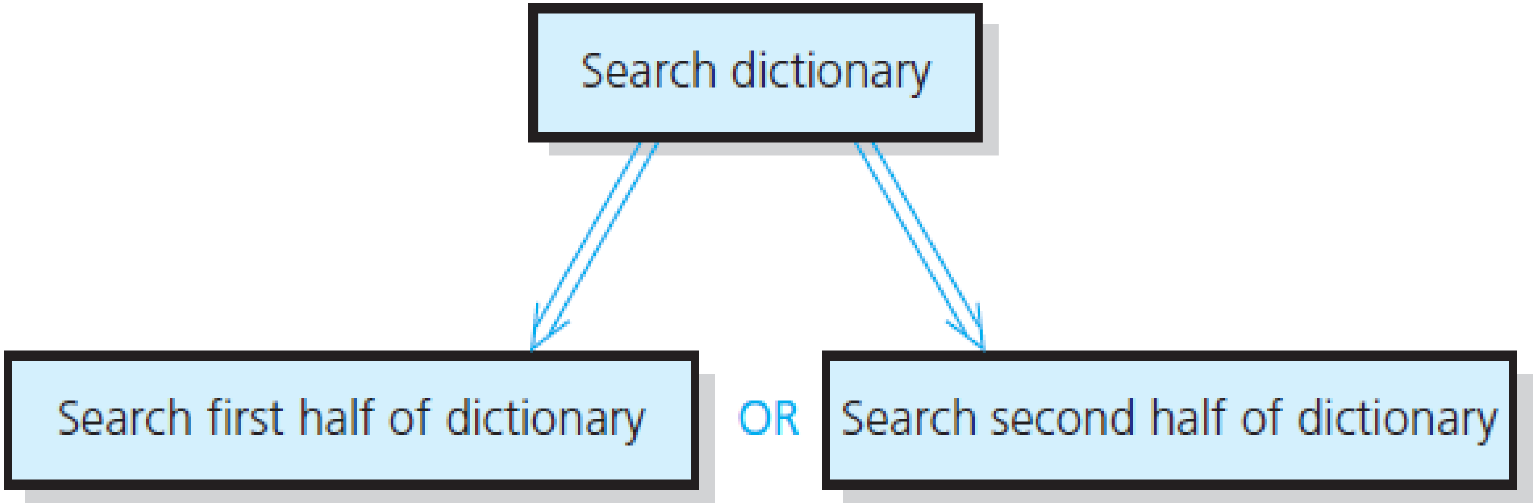
Recursive Solutions

- Recursion breaks a problem into smaller identical problems

- Some recursive solutions are inefficient, impractical

- Complex problems can have simple recursive solutions

Recursive Solutions



What Is Recursion?

- **Recursive call:** A method call in which the method being called is the same as the one making the call

- **Direct recursion:** Recursion in which a method directly calls itself

- **Indirect recursion:** Recursion in which a chain of two or more method calls returns to the method that originated the chain

Recursion

- You must be careful when using recursion.
- Recursive solutions are typically less efficient than iterative solutions. **Avoid them !!!**
- Still, many problems lend themselves to simple, elegant, recursive solutions.
- We must avoid making an infinite sequence of function calls
  - infinite recursion

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

Recursive Solutions

Questions for constructing recursive solutions

1. How to define the problem in terms of a smaller problem of same type?
2. How does each recursive call diminish the size of the problem?
3. What instance of problem can serve as base case?
4. As problem size diminishes, will you reach base case?

A Recursive Valued Function: The Factorial of *n*

- An iterative solution

$$factorial(n) = n \times (n-1) \times (n-2) \times \cdots \times 1 \quad \text{for an integer } n > 0$$
$$factorial(0) = 1$$

- A factorial solution

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

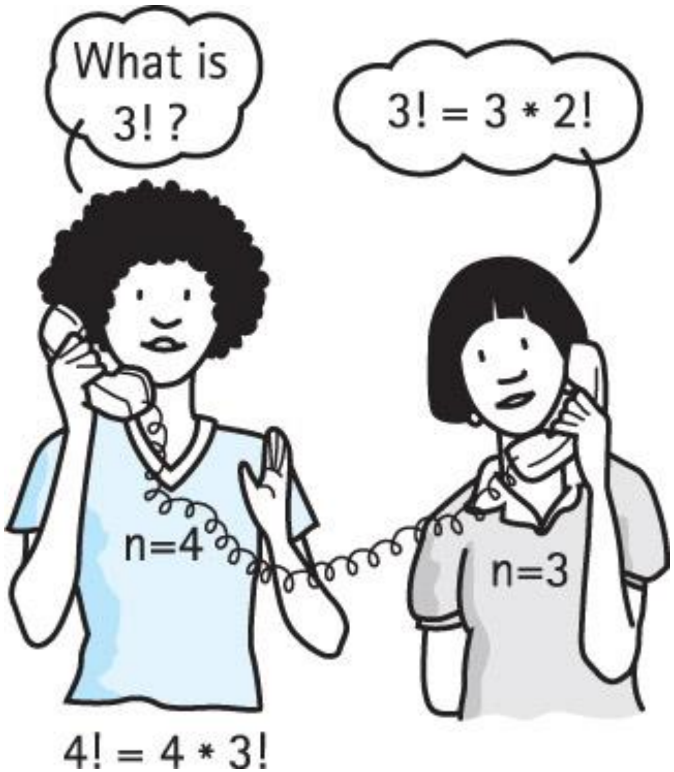
Note: Do not use recursion if a problem has a simple, efficient iterative solution

General format for many recursive functions

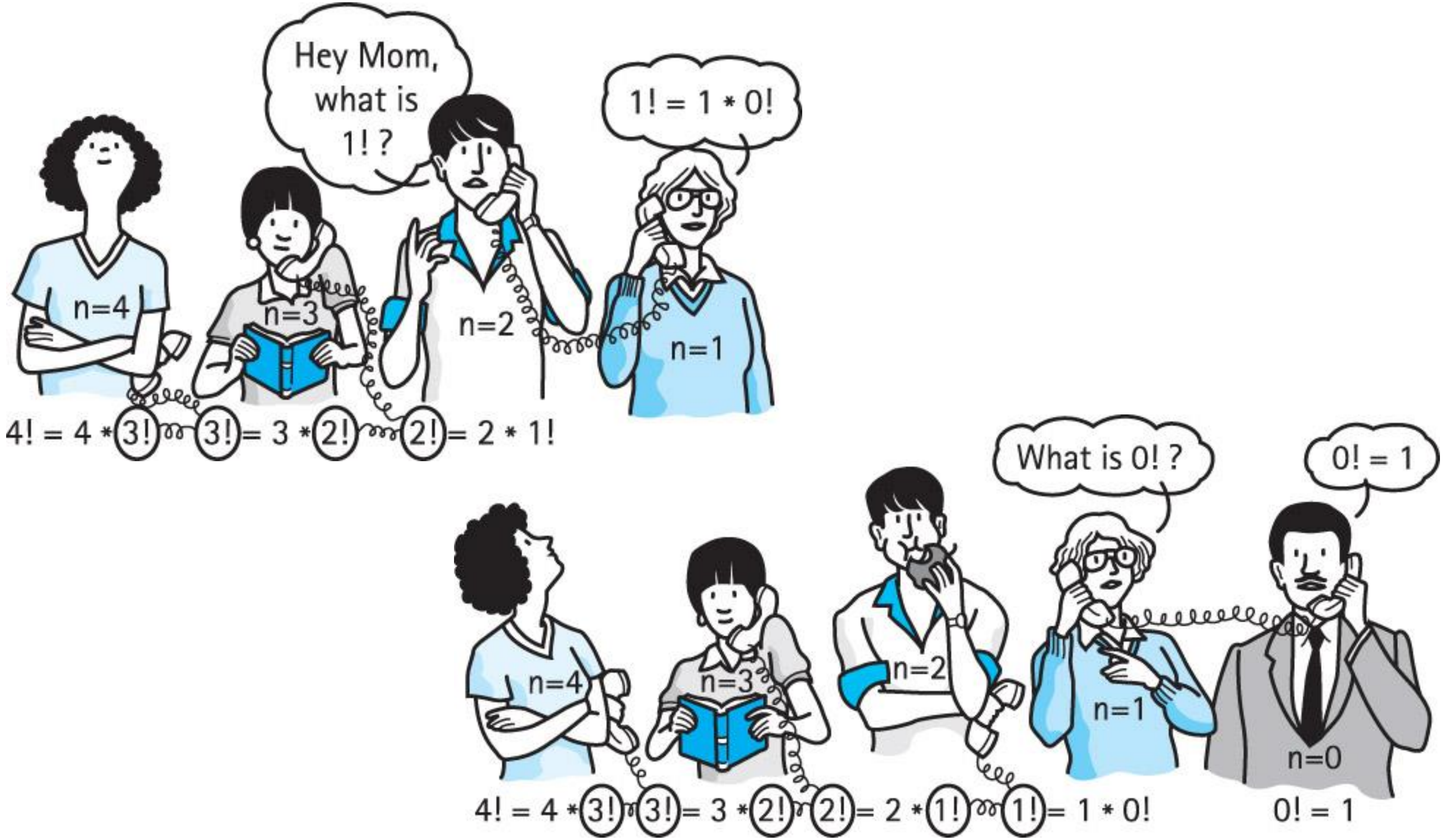
```
if   (some condition for which answer is known)
                                          // base case
    solution statement
else                                      // general case
    recursive function call
```

- Each successive recursive call should bring you closer to a situation in which the answer is known.
- Each recursive algorithm must have at least one base case, as well as the general (recursive) case

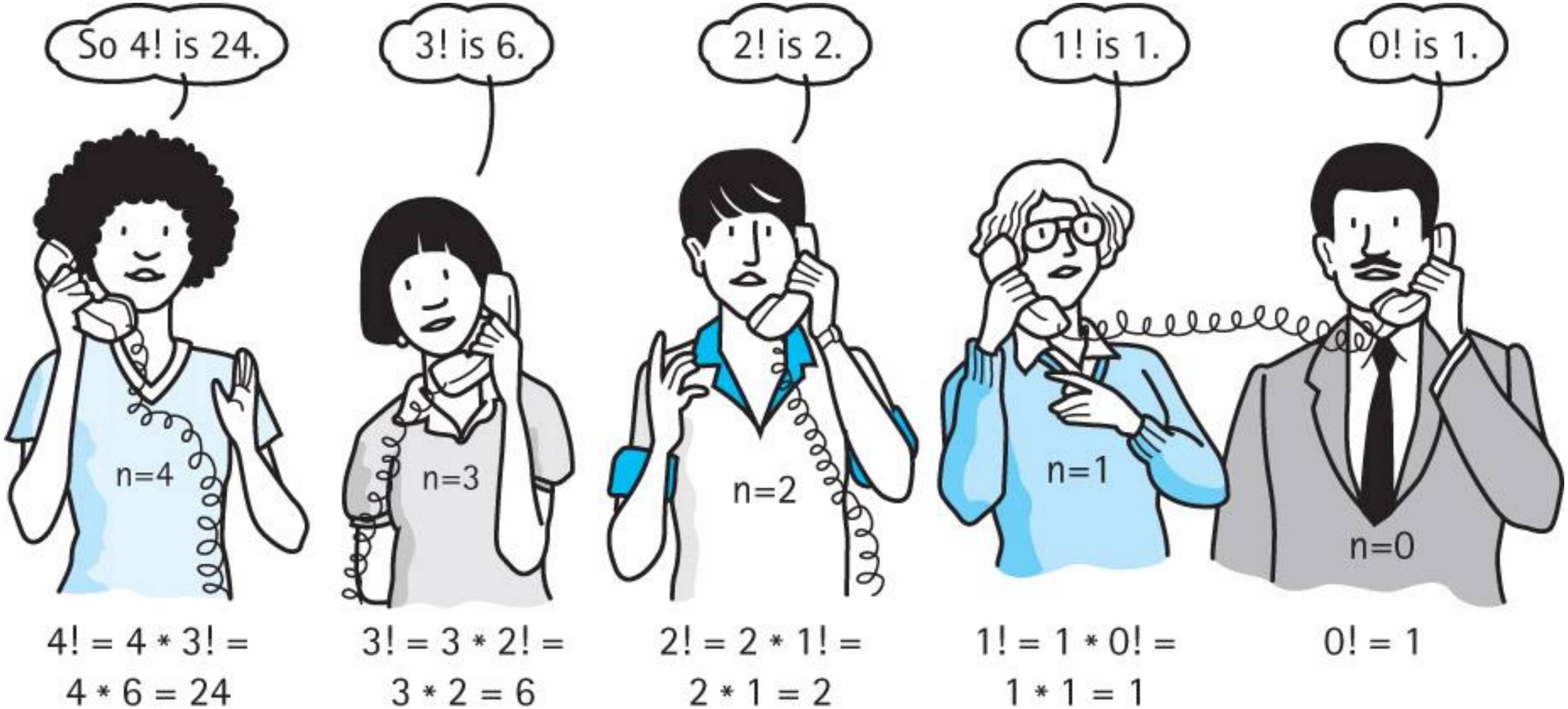
Recursive Query



Recursive Query



Recursive Solution



A Recursive Valued Function

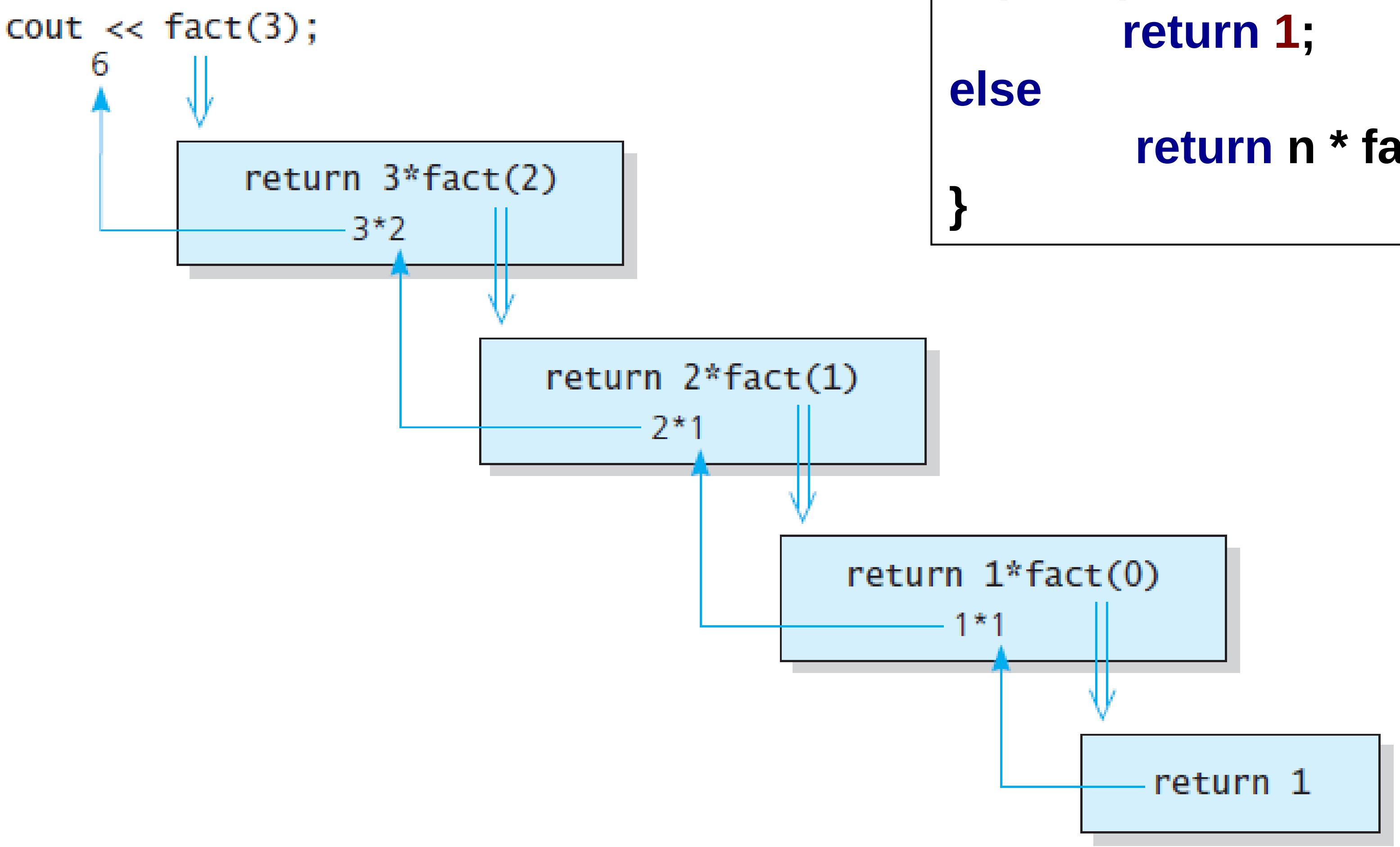
The factorial of n

```
/** 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

```
int fact(int n)
{
 if(n==0)
           return 1;
else
           return n * fact(n-1);
}
```

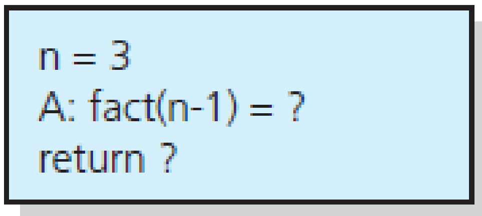
fact(3)

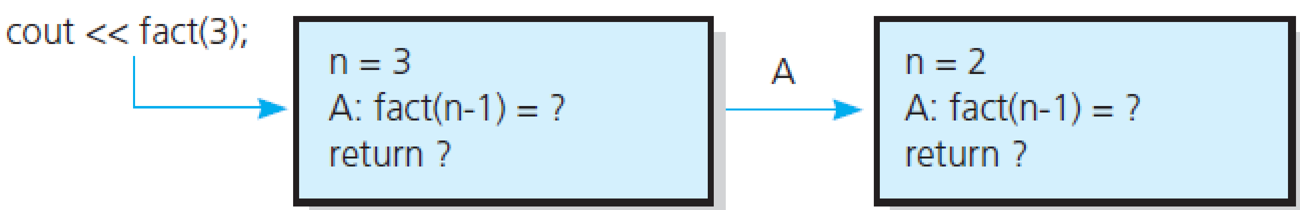


The Box Trace

1. Label each recursive call
2. Represent each call to function by a new box
3. Draw arrow from box that makes call to newly created box
4. After you create new box executing body of function
5. On exiting function, cross off current box and follow its arrow back

The Box Trace

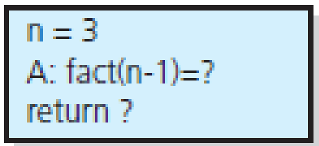




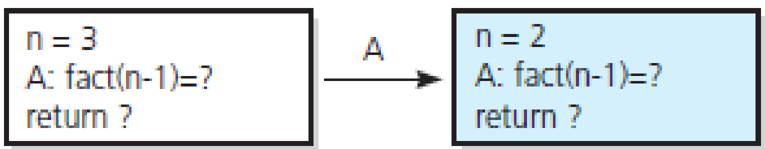
The beginning of the box trace

The Box Trace

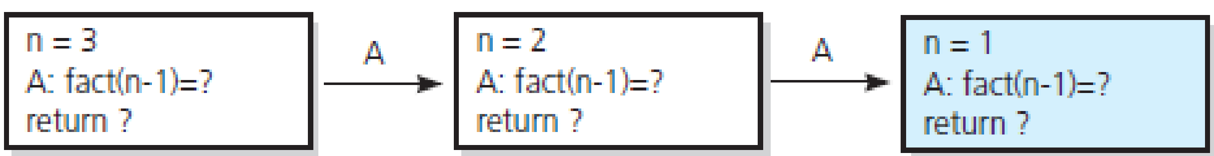
The initial call is made, and method `fact` begins execution:



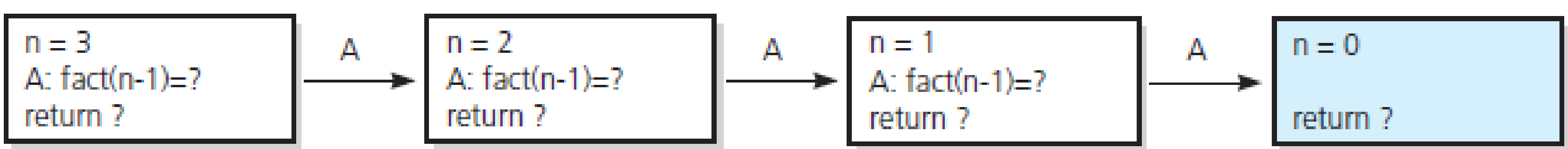
At point A a recursive call is made, and the new invocation of the method `fact` begins execution:



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

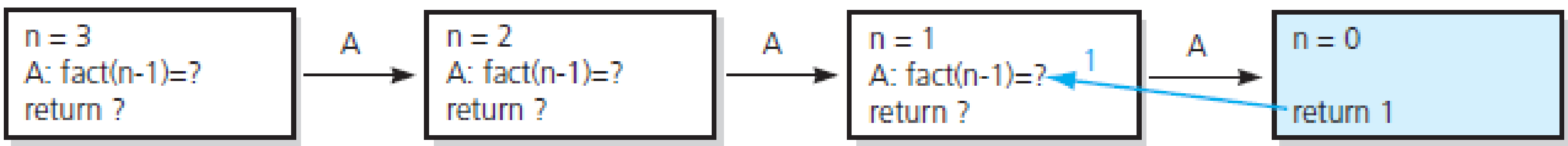


At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

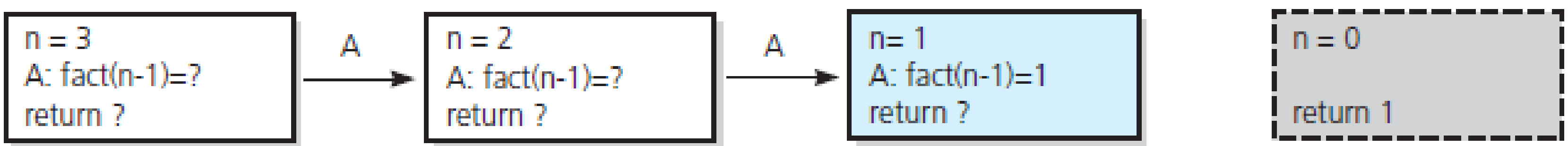


The Box Trace

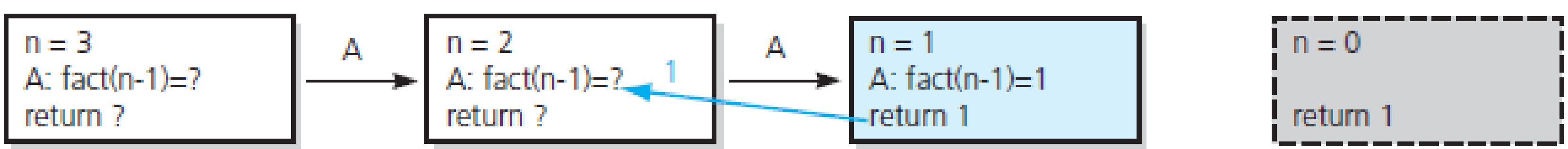
This is the base case, so this invocation of `fact` completes and returns a value to the caller:



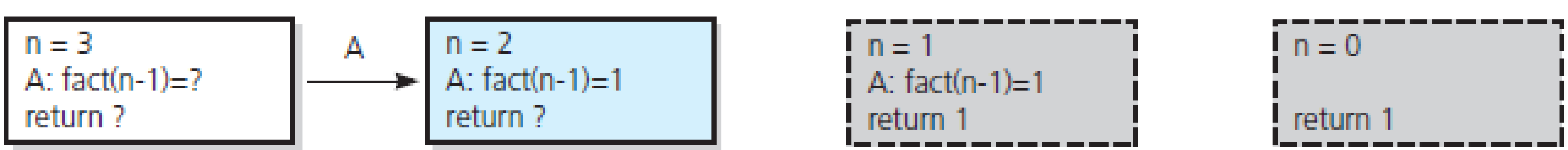
The method value is returned to the calling box, which continues execution:



The current invocation of `fact` completes and returns a value to the caller:



The method value is returned to the calling box, which continues execution:

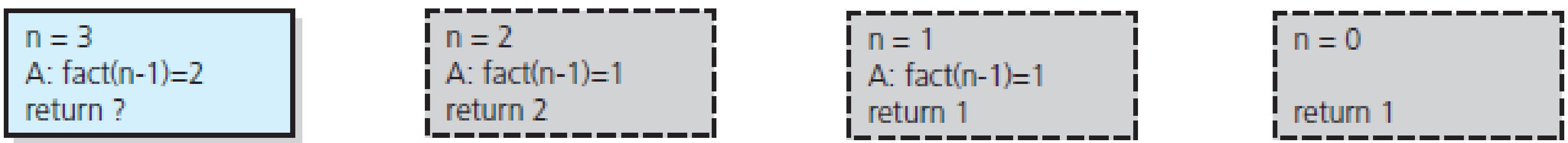


The Box Trace

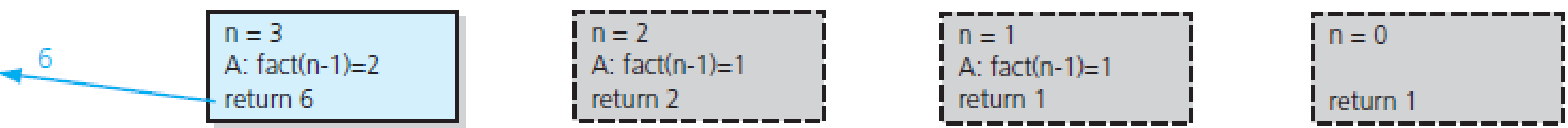
The method value is returned to the calling box, which continues execution:

The current invocation of `fact` completes and returns a value to the caller:

The method value is returned to the calling box, which continues execution:



The current invocation of `fact` completes and returns a value to the caller:



The value 6 is returned to the initial call.

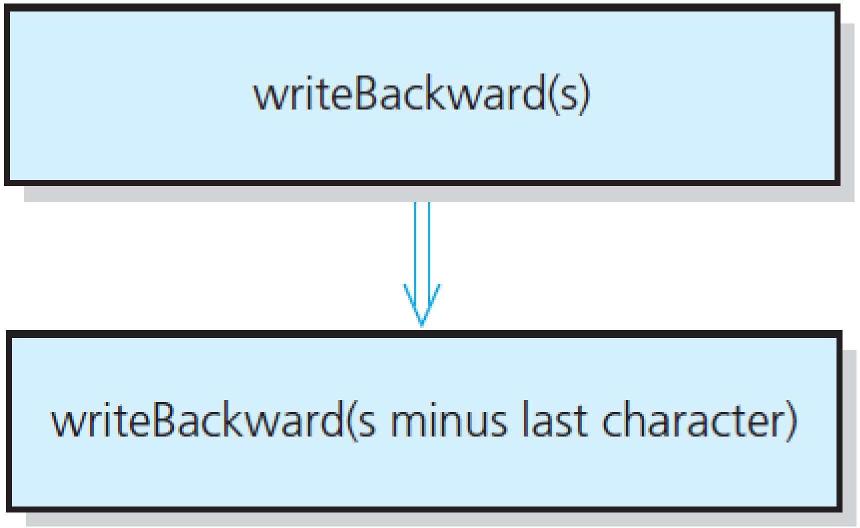
Designing a recursive solution: Writing a String Backward

- **Problem:**
  - *Given a string of characters, write it in reverse order*
- **Recursive solution:**
  - How can the problem be defined in terms of smaller problems of the same type?
    - We could write the last character of the string and then solve the problem of writing first n-1 characters backward
  - By how much does each recursive call reduce the problem size?
    - Each recursive step of the solution diminishes by 1 the length of the string to be written backward
  - What is the base case that can be solved without recursion?
    - Base case: Write the empty string backward = Do nothing.
  - Will the base case be reached as the problem size is reduced?
    - Yes.

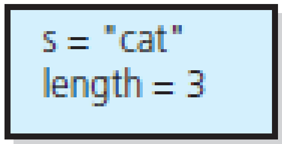
Designing a recursive solution: Writing a String Backward

```
void writeBackward(String s) {
/** Writes a character string backward.
  * @pre: The string s contains length characters, where length >= 0.
  * @post: s is written backward, but remains unchanged.
  */

  int length = s.size();

  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



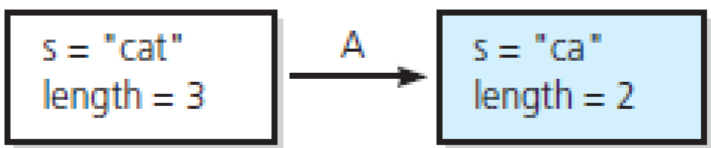
A Recursive Void Function



Output line: **t**

Point A (`writeBackward(s)`) is reached, and the recursive call is made.

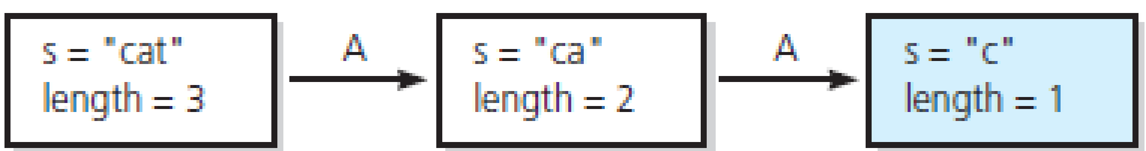
The new invocation begins execution:



Output line: **ta**

Point A is reached, and the recursive call is made.

The new invocation begins execution:



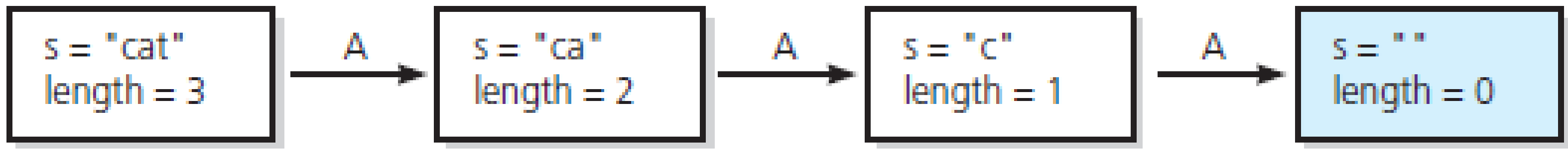
Box trace of **`writeBackward("cat")`**

A Recursive Void Function

Output line: **tac**

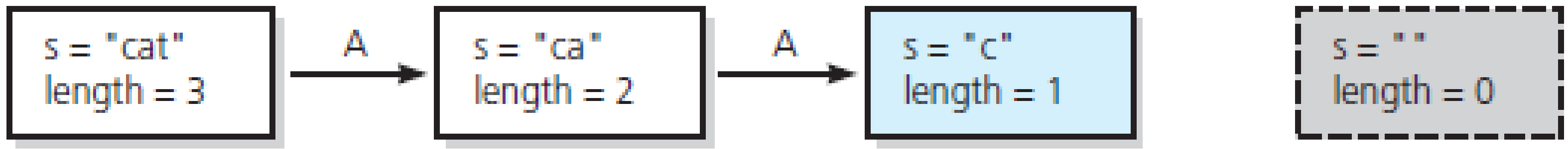
Point A is reached, and the recursive call is made.

The new invocation begins execution:



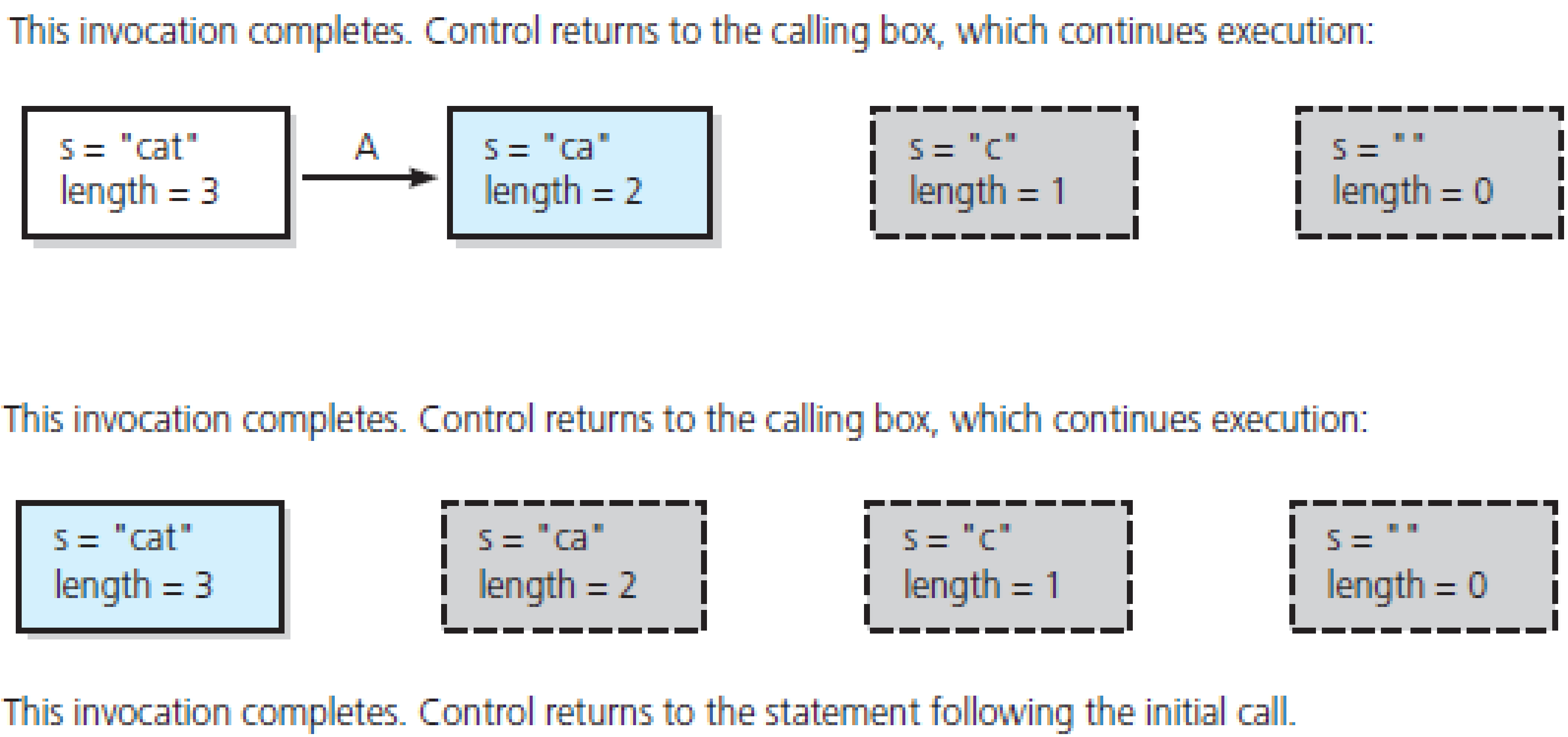
This is the base case, so this invocation completes.

Control returns to the calling box, which continues execution:



Box trace of **`writeBackward("cat")`**

A Recursive Void Function



Box trace of **`writeBackward("cat")`**

Writing an Array's Entries in Backward Order

```
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)
   }
```

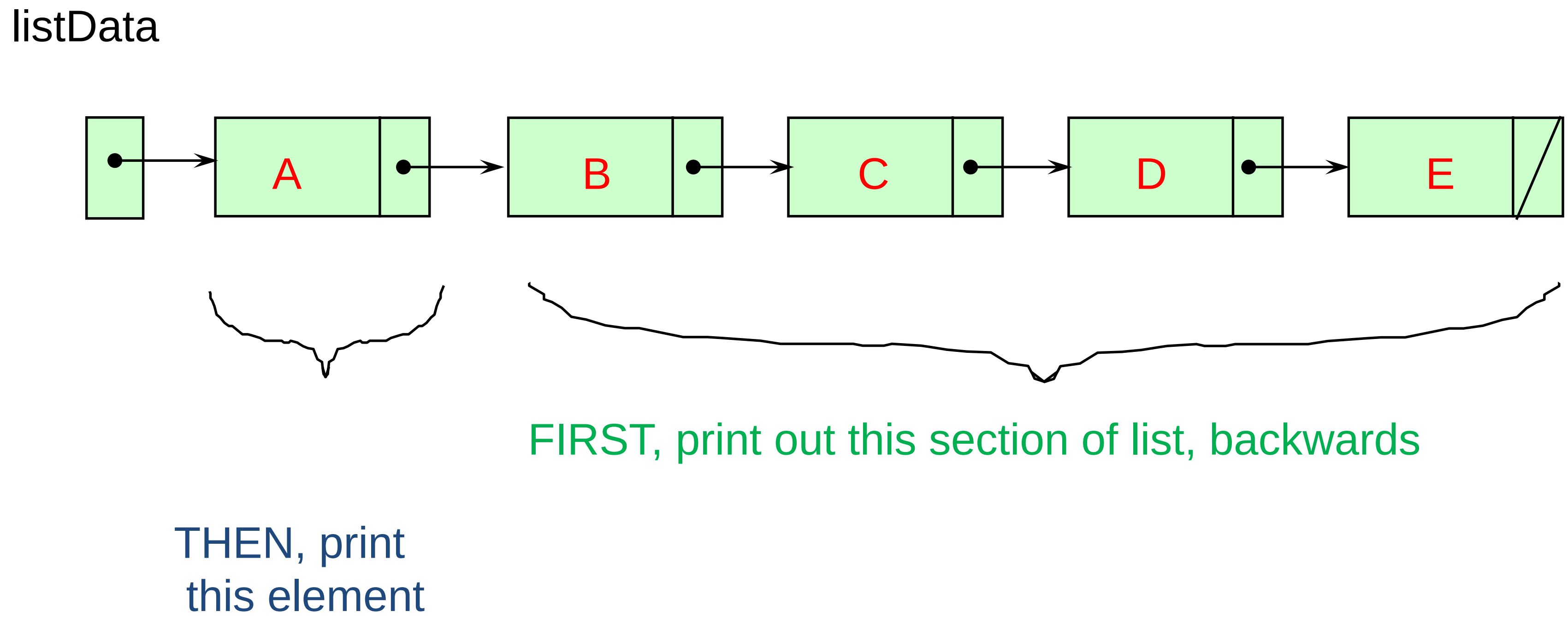
Pseudocode

Writing an Array's Entries in Backward Order

```
/** 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
```

Source code

RevPrint(listData);



Using recursion with a linked list

```
void   RevPrint ( NodeType*  listPtr )

/**  Reverse print a linked list
   @Pre listPtr points to an element of a list.
   @Post  all elements of list pointed to by listPtr
   have been printed out in reverse order.   **/
{
    if  ( listPtr != NULL )          // general case
    {
      RevPrint ( listPtr-> next ); // process the rest
      std::cout << listPtr->info << std::endl;
                                   // print this element
    }
   // Base case : if the list is empty, do nothing
}
```

The Binary Search

```
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)
   }
```

A high-level binary search for the array problem

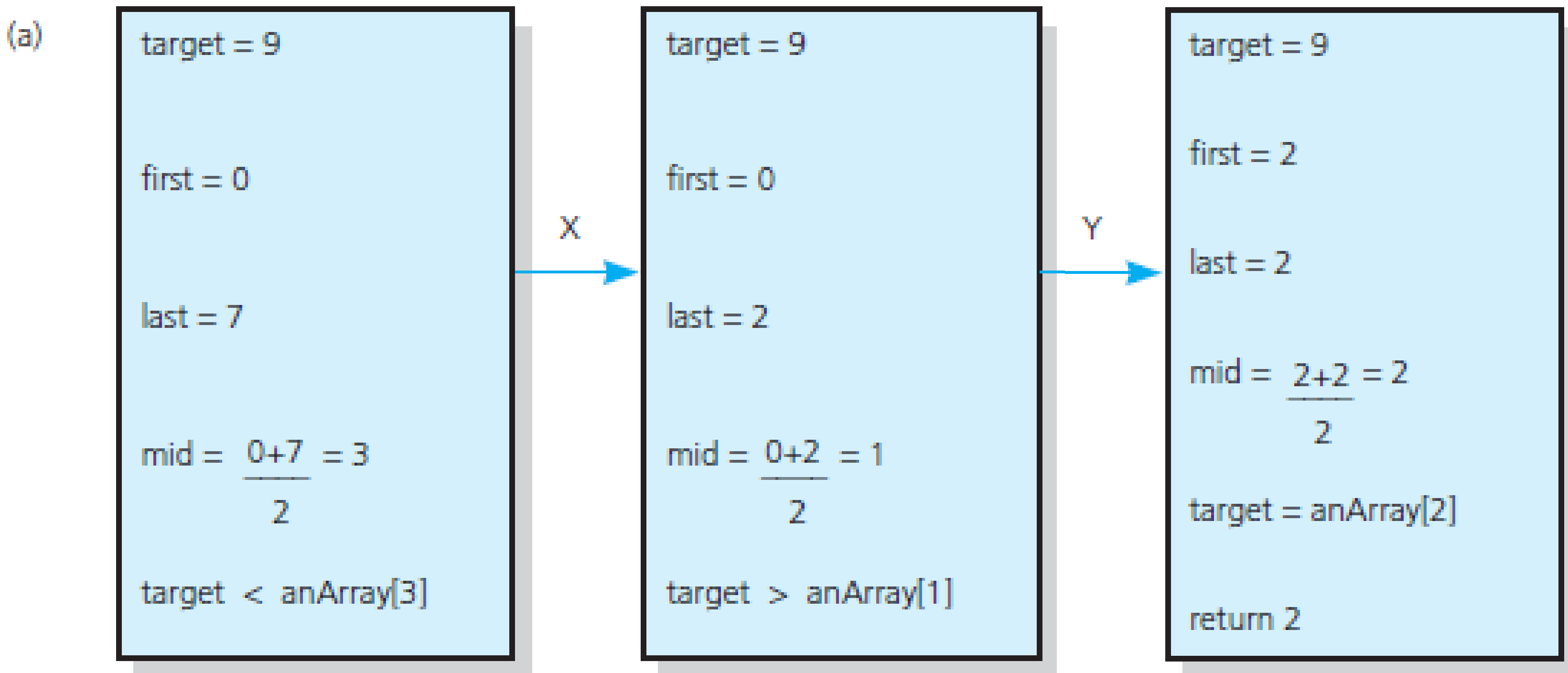
The Binary Search

Consider details before implementing algorithm:

1. How to pass half of anArray to recursive calls of binarySearch ?
2. How to determine which half of array contains target?
3. What should base case(s) be?
4. How will binarySearch indicate result of search?

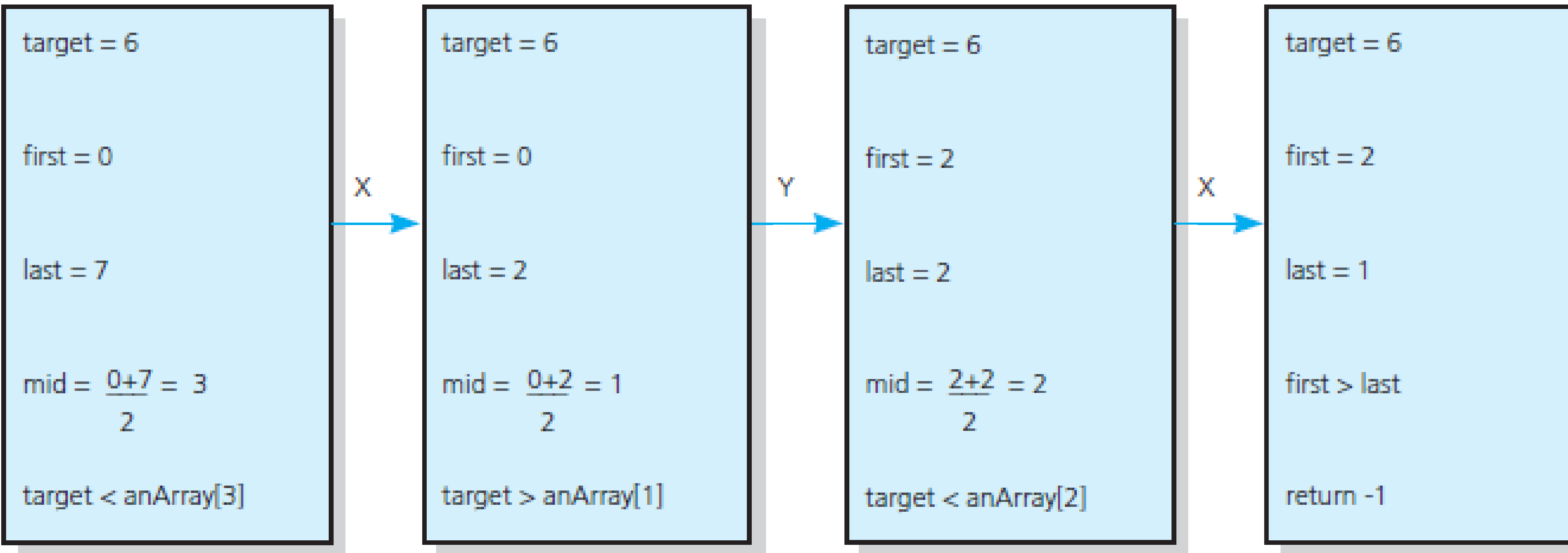
```
/** A recursive binary search function.
  * @return: It returns location of x in given array
  * arr[l..r] is present, otherwise -1
  */
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l){
        int mid = l + (r - l)/2;
        // If the element is present at the middle itself

        if (arr[mid] == x)
            return mid;
        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid-1, x);
        // Else the element can only be present in right subarray
        return binarySearch(arr, mid+1, r, x);
    }
// We reach here when element is not present in array
return -1;
}
```

The Binary Search



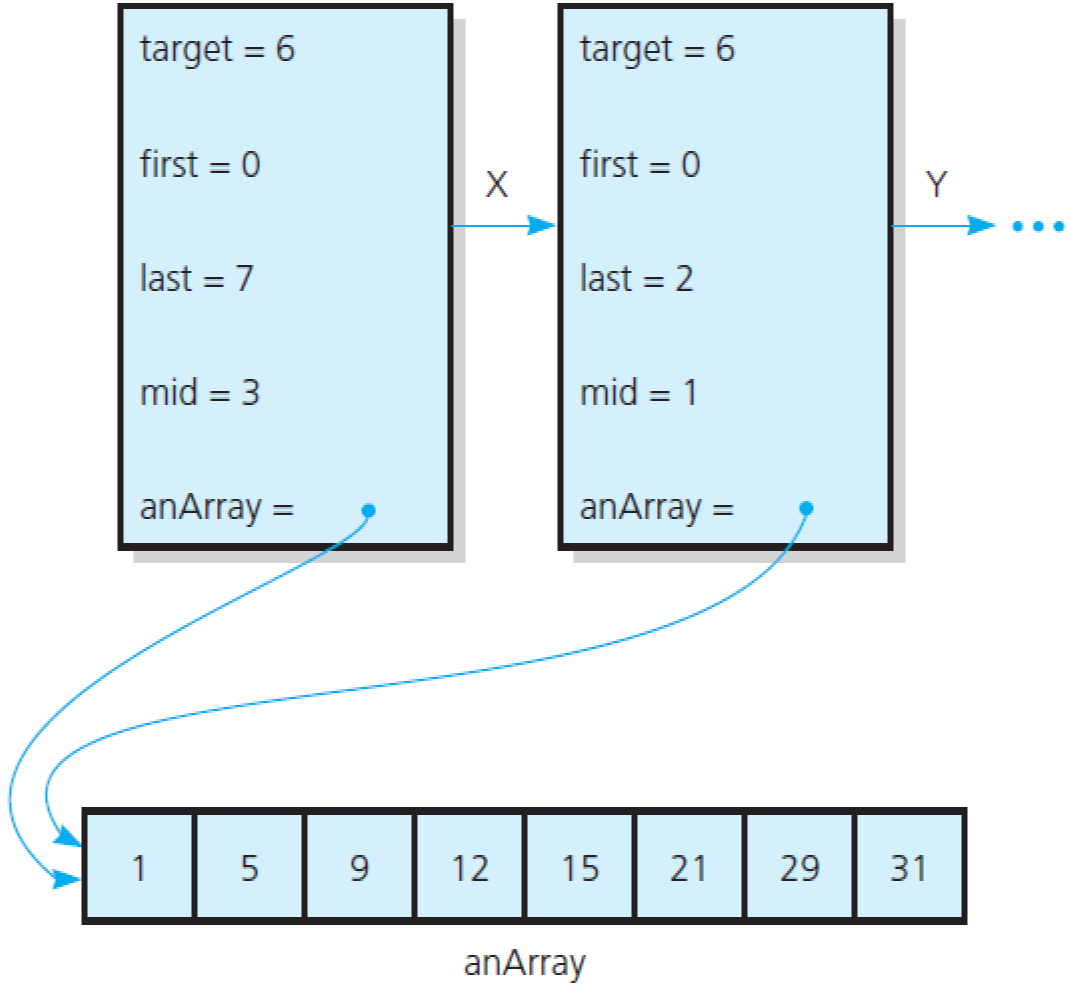
Box traces of **`binarySearch`** with **`anArray`** = <1, 5, 9, 12, 15, 21, 29, 31>:
(a) a successful search for 9

The Binary Search



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

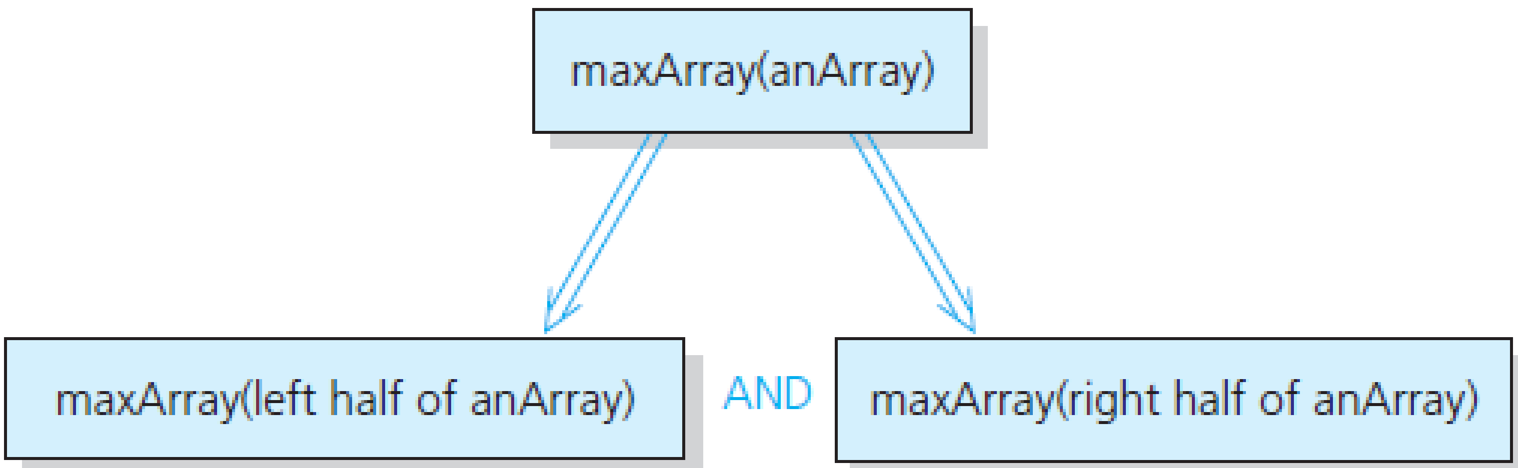
The Binary Search



Box trace with a reference argument

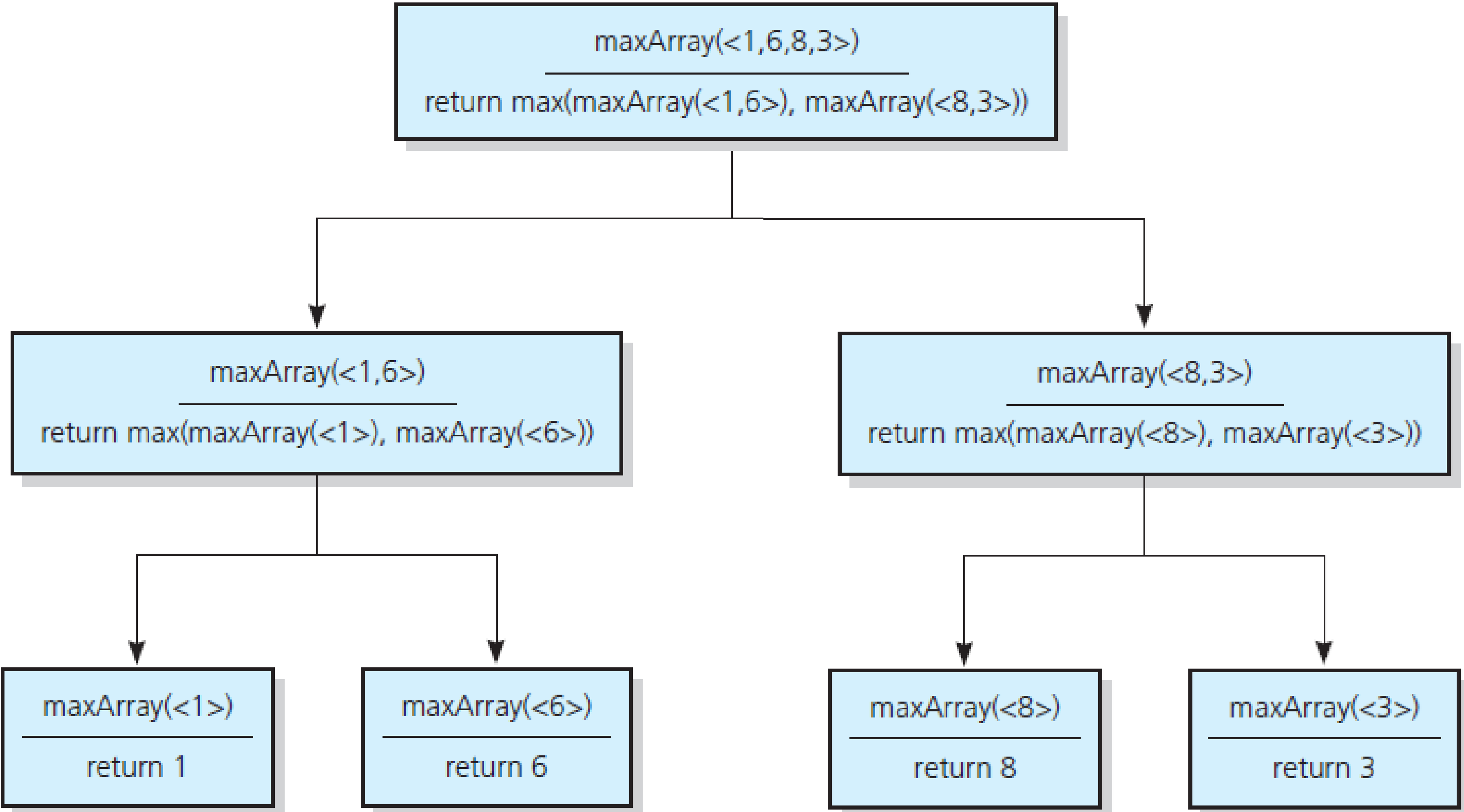
Finding the Largest Value in an Array

```
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)
```



Recursive solution to the largest-value problem

Finding the Largest Value in an Array

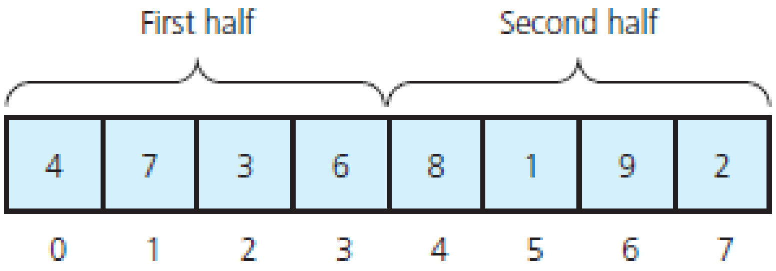


The recursive calls that **`maxArray(<1,6,8,3>)`** generates

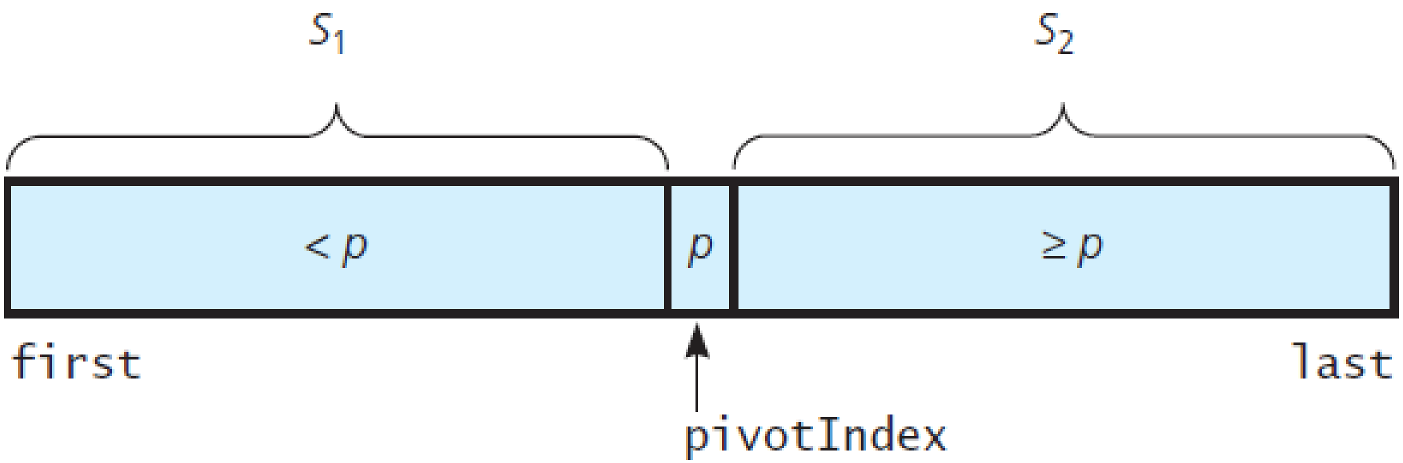
Finding the k^{th} Smallest Value of an Array

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



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



A partition about a pivot

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

```
// 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)
```

High level pseudo code solution

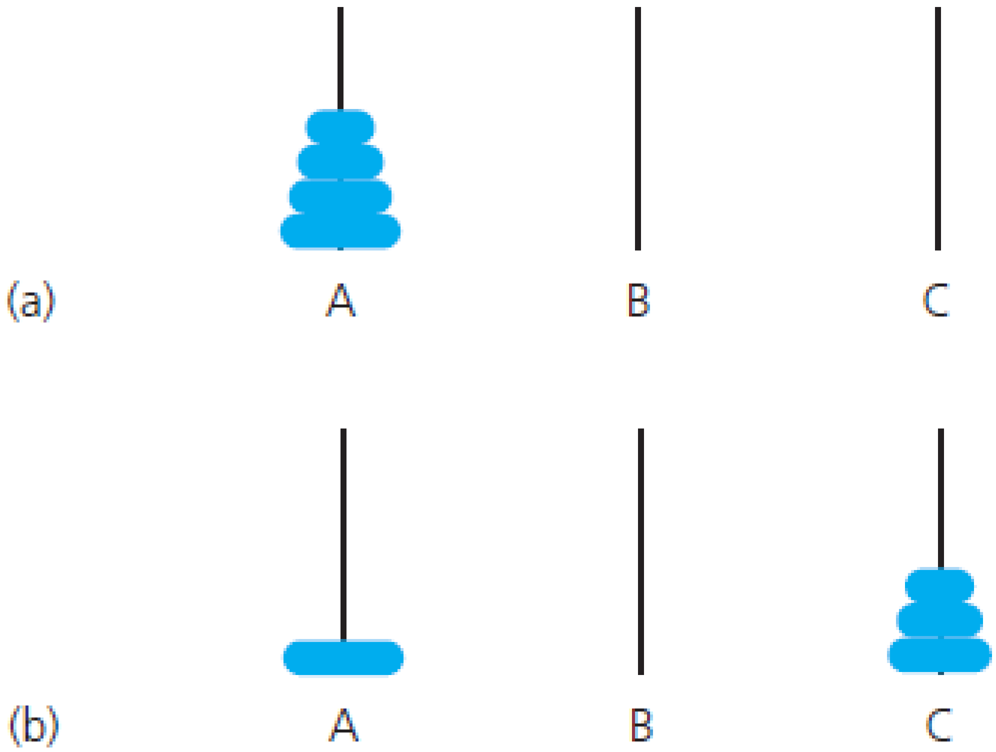
The Towers of Hanoi

- a mathematical game or puzzle 
  - consists of three rods and a number of disks of different sizes, which can slide onto any rod.
- The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
  1. Only one disk can be moved at a time.
  2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack.
  3. No disk may be placed on top of a smaller disk.

The Towers of Hanoi

- The problem statement
  - Beginning with *n* disks on pole A and zero disks on poles B and C, solve towers(n, A, B, C) .
- Solution
  1. With all disks on A, solve towers(n – 1, A, C, B)
  2. With the largest disk on pole A and all others on pole C, solve towers(n – 1, A, B, C)
  3. With the largest disk on pole B and all the other disks on pole C, solve towers(n – 1, C, B, A)

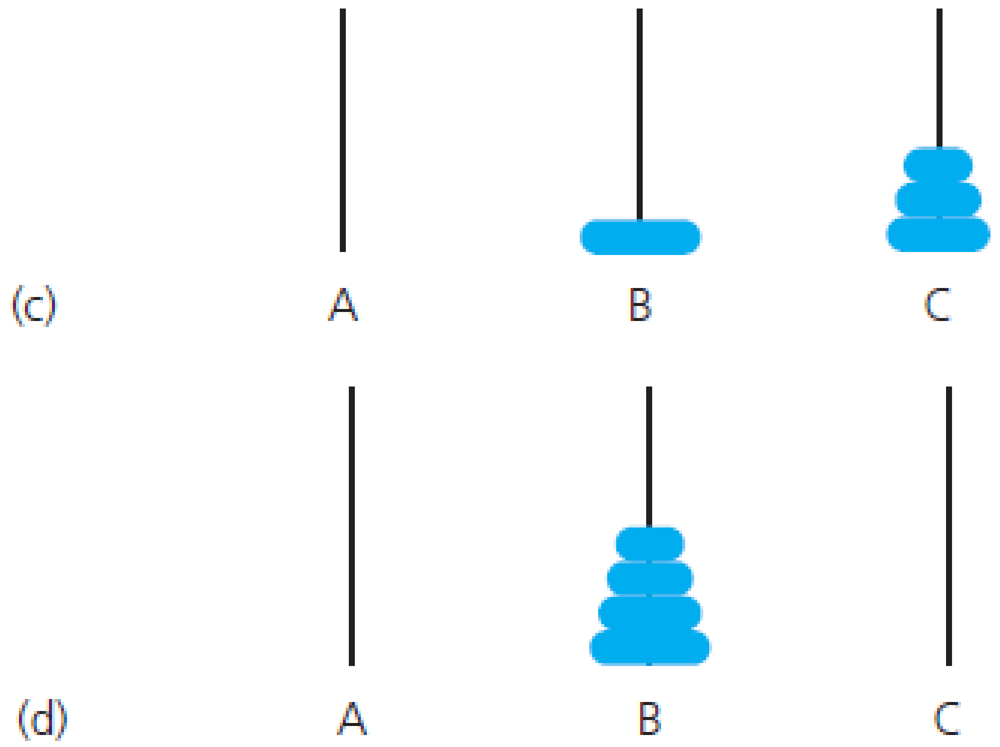
Organizing Towers of Hanoi



(a) the initial state;

(b) move n – 1 disks from A to C;

Towers of Hanoi

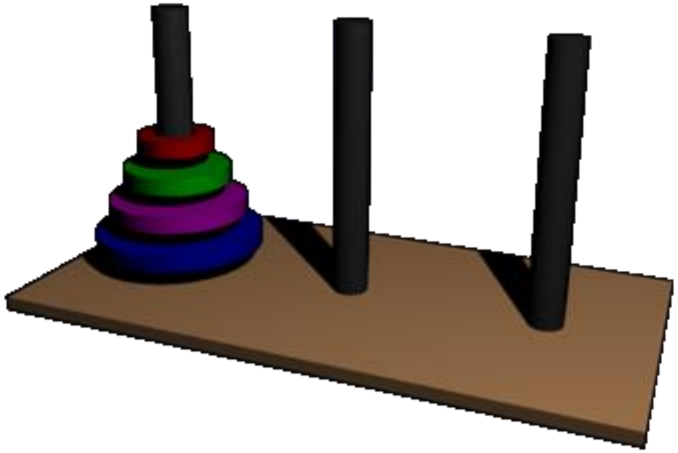


(c) move 1 disk from A to B;

(d) move n – 1 disks from C to B

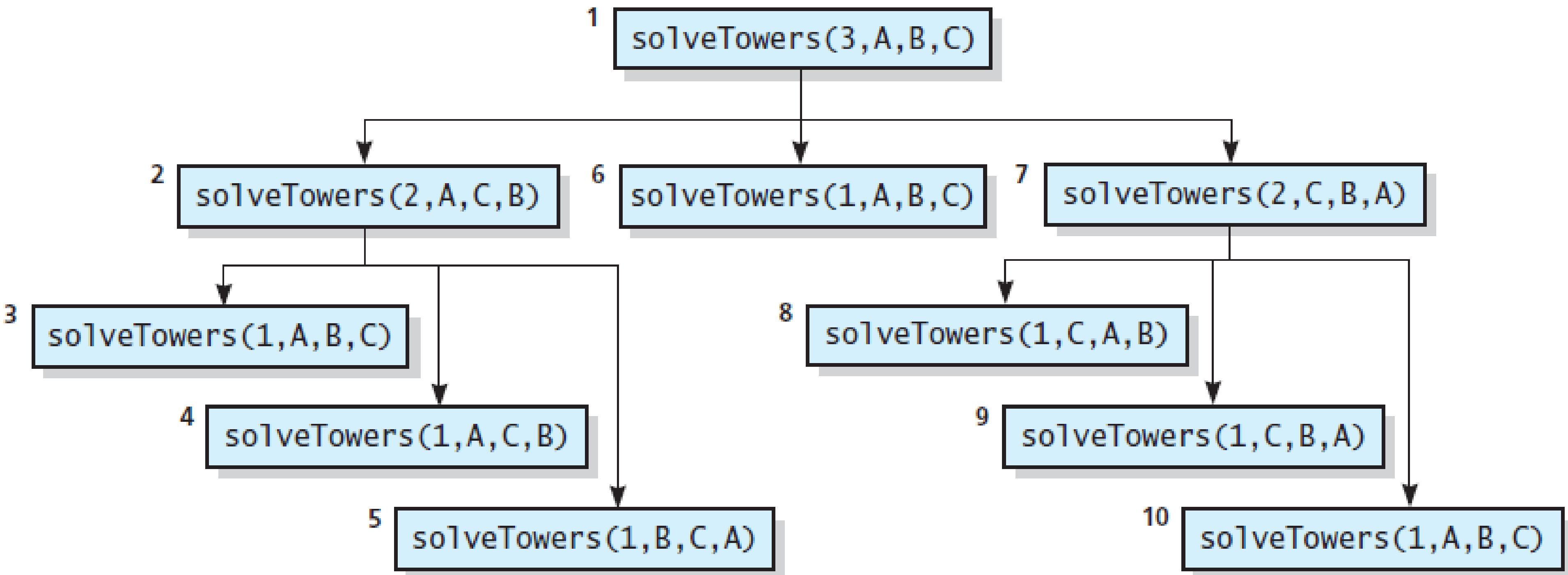
Towers of Hanoi

```
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)
    }
```



Pseudocode solution

Towers of Hanoi



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

Towers of Hanoi

```
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
```

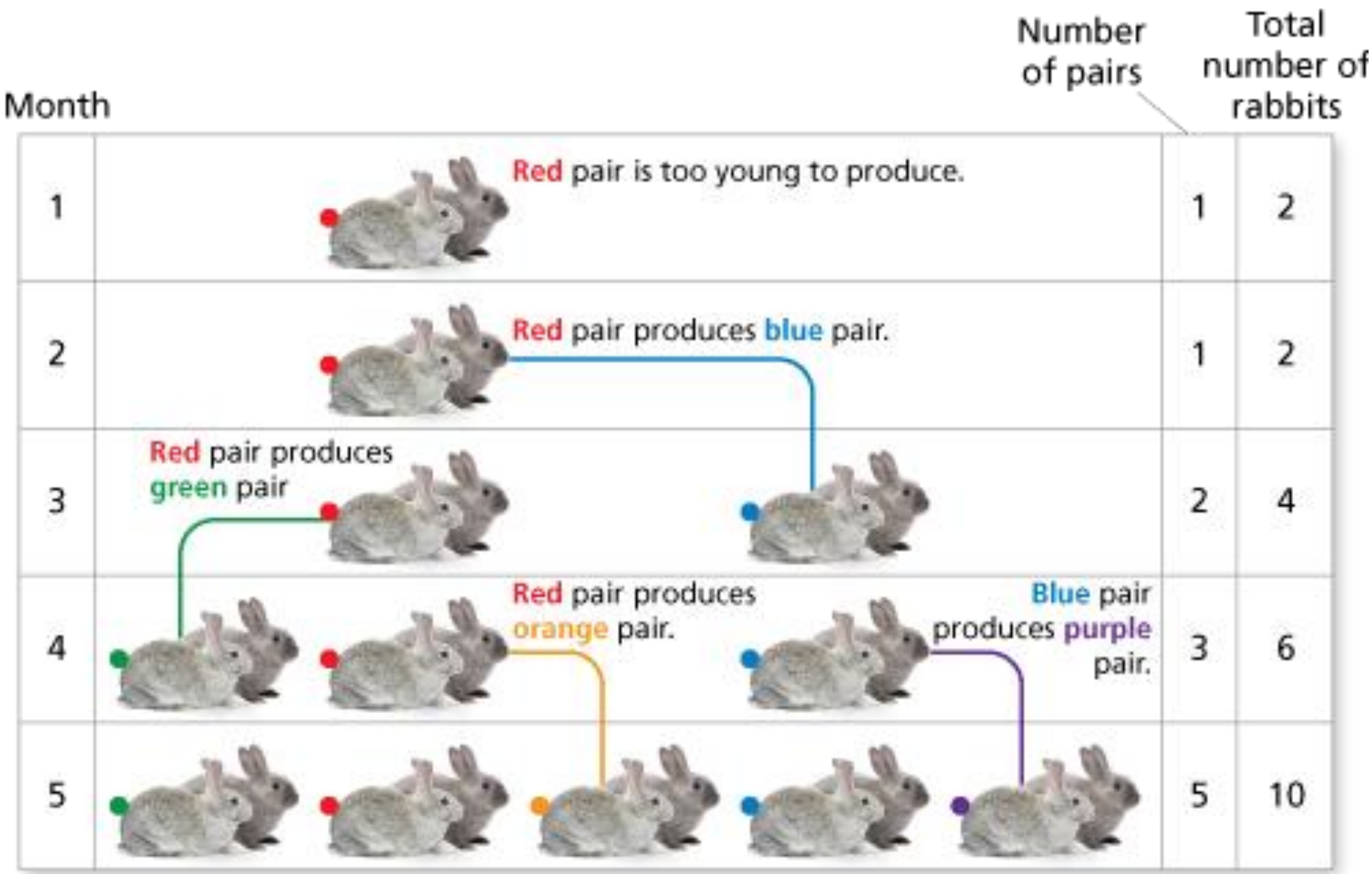
Source code for **`solveTowers`**

Multiplying Rabbits

Assumed “facts” about rabbits:

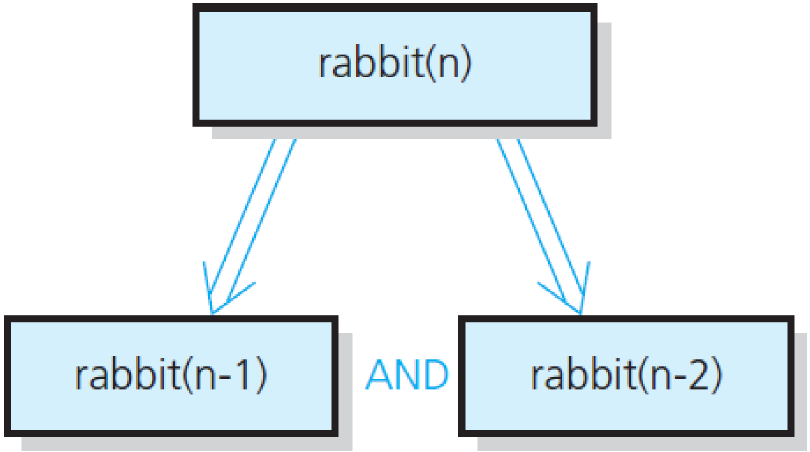
- Rabbits never die.
- A rabbit reaches maturity exactly two months after birth
- Rabbits always born in male-female pairs.
- At the beginning of every month, each mature male-female pair gives birth to exactly one male-female pair.

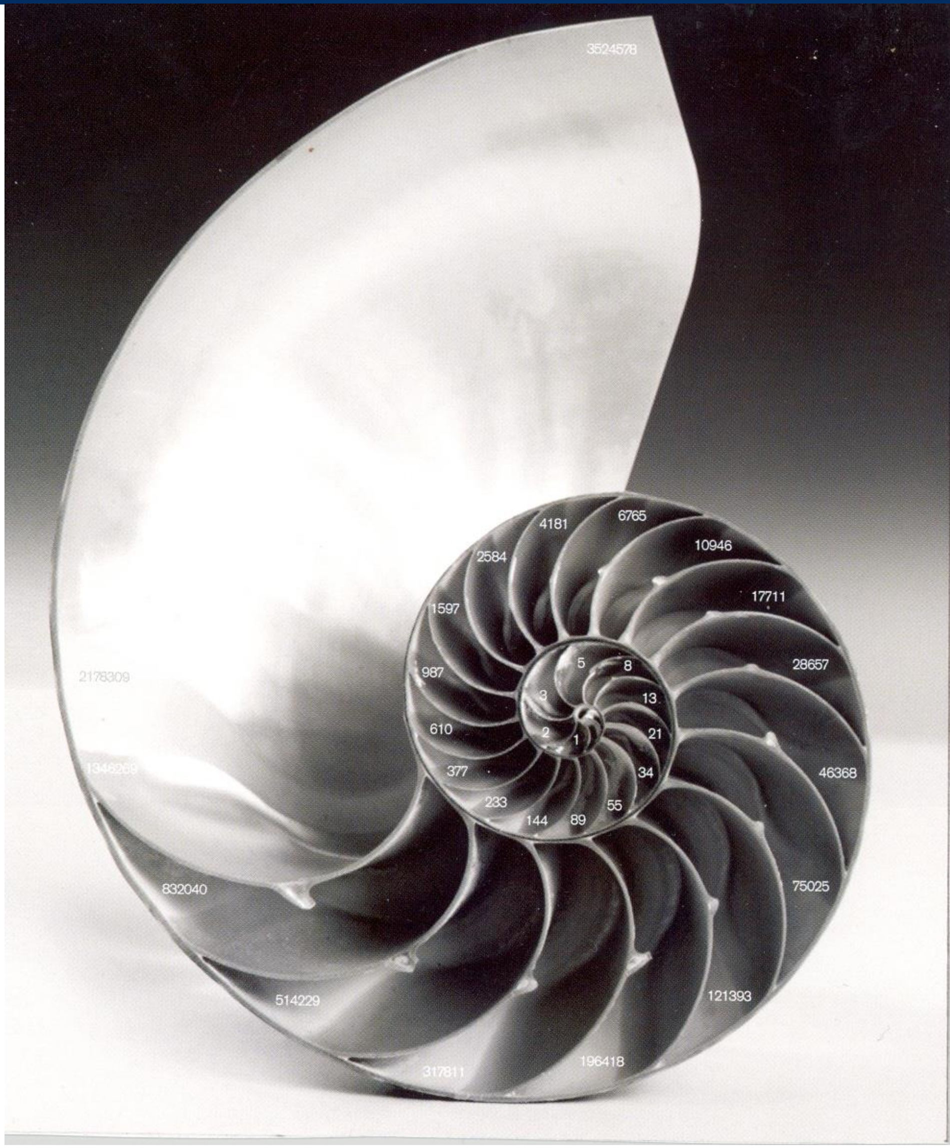
Multiplying Rabbits



Multiplying Rabbits (The Fibonacci Sequence)

to find the next number in the sequence,
add together the previous two numbers



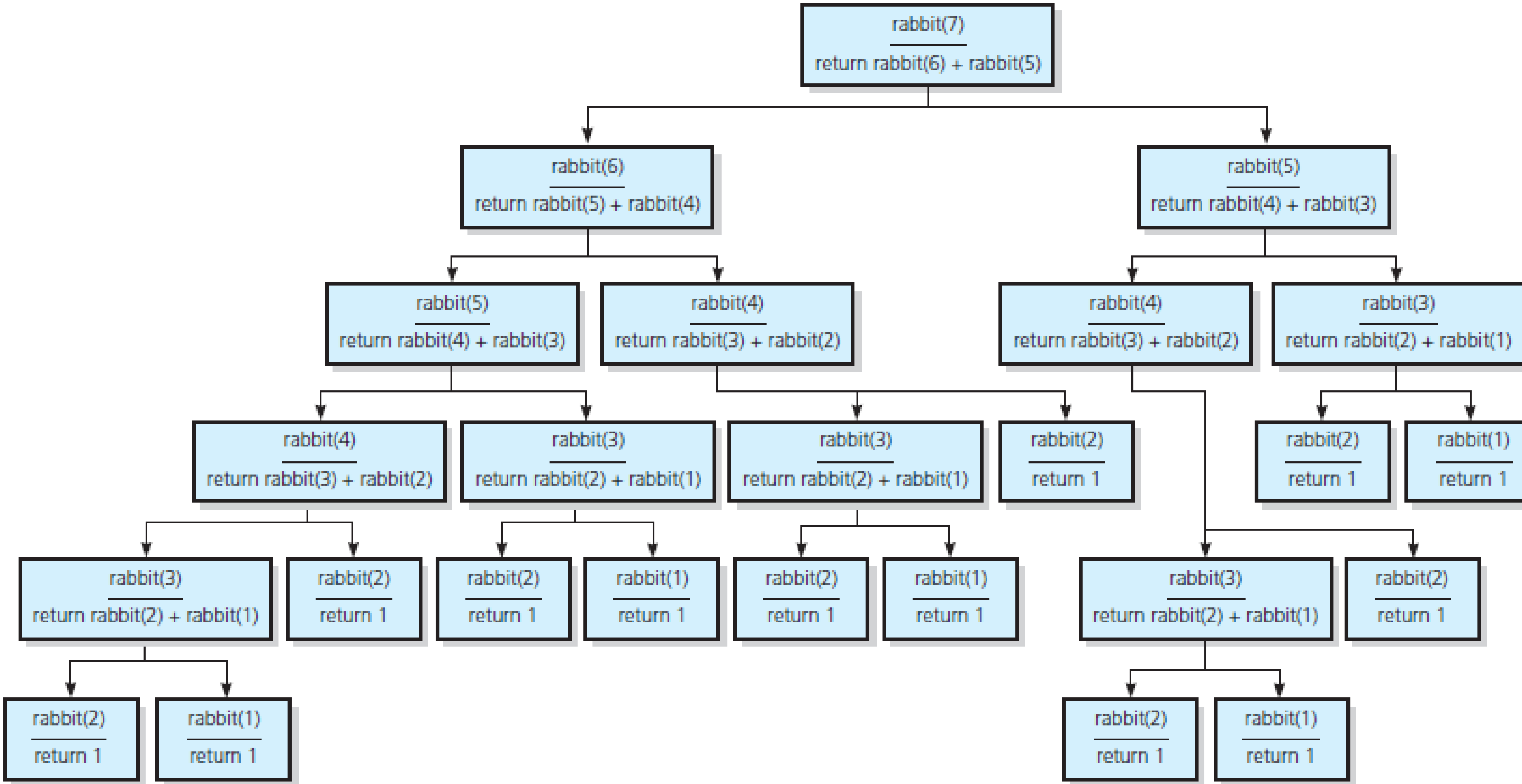


The Fibonacci Sequence (Multiplying Rabbits)

```
/** 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
```

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

The Fibonacci Sequence (Multiplying Rabbits)



The recursive calls that rabbit(7) generates

Organizing a Parade

- Will consist of bands and floats in single line.
  - You are asked not to place one band immediately after another
- In how many ways can you organize a parade of length n ?
  - $P(n)$ = number of ways to organize parade of length n
  - $F(n)$ = number of parades of length n, end with a float
  - $B(n)$ = number of parades of length n, end with a band
- Then $P(n) = F(n) + B(n)$

Organizing a Parade

- $F(n) = P(n-1)$
- $B(n) = F(n-1) = P(n-2)$

- $P(n) = P(n - 1) + P(n - 2)$ for $n > 2$
- $P(1) = 2$
- $P(2) = 3$

- Thus a recursive solution
  - Solve the problem by breaking up into cases

Choosing *k* Out of *n* Things

- Rock band wants to tour *k* out of *n* cities
  - Order not an issue
- Let *g(n, k)* be number of groups of *k* cities chosen from *n*

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

- *Base cases*

$$g(k,k) = 1$$
$$g(n,0) = 1$$

Combinations

- how many combinations of a certain size can be made out of a total group of elements

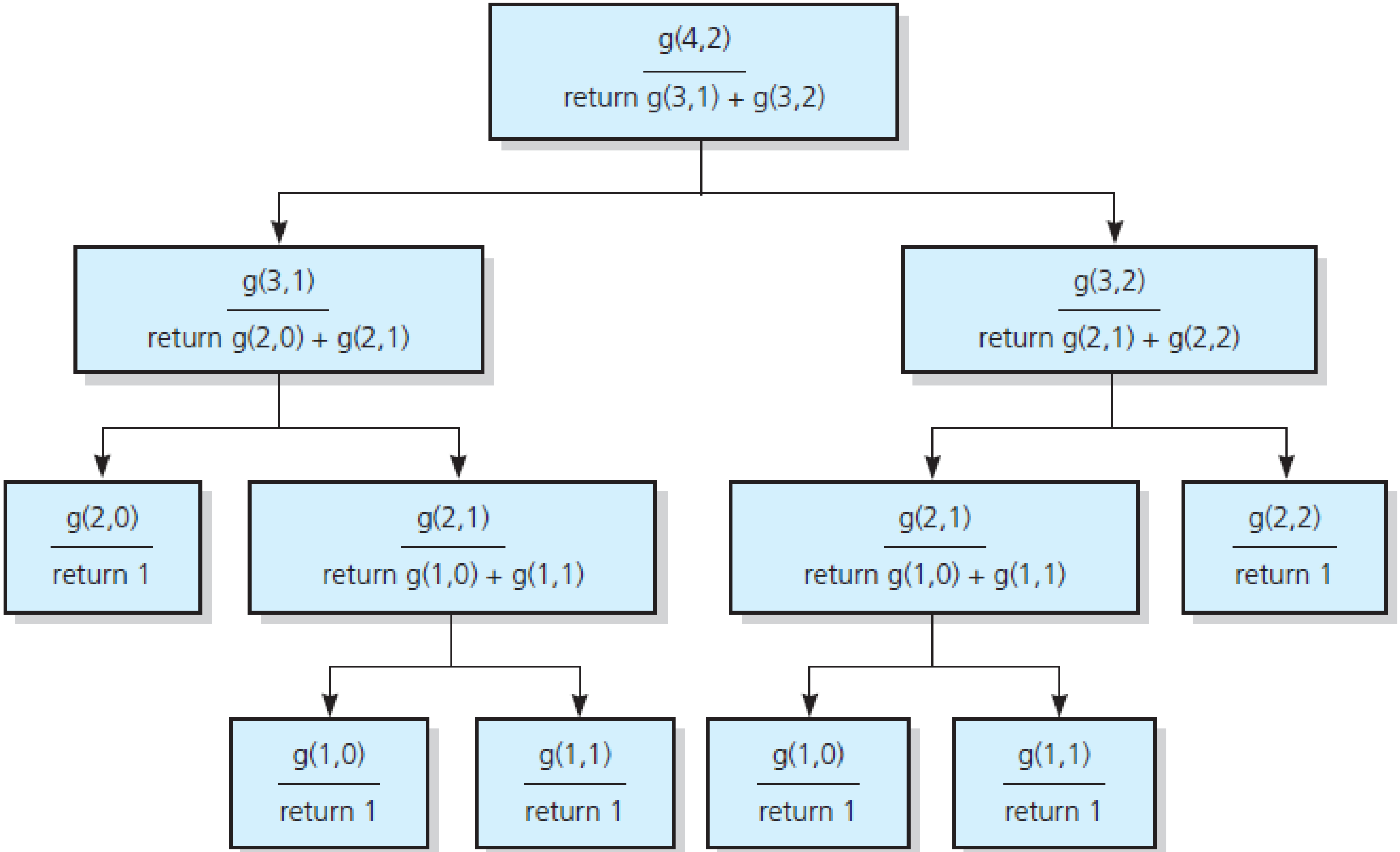
$$g(n,k) = \begin{cases} 1 & \text{if } k = 0 \\ 1 & \text{if } k = n \\ 0 & \text{if } k > n \\ g(n-1,k-1) + g(n-1,k) & \text{if } 0 < k < n \end{cases}$$

Choosing *k* Out of *n* Things

```
/** 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
```

Recursive function:

Choosing *k* Out of *n* Things



The recursive calls that g (4, 2) generates

Tail Recursion

- The case in which a function contains only a single recursive call and it is the last statement to be executed in the function.

- Tail recursion can be replaced by iteration to remove recursion from the solution

```
// USES TAIL RECURSION
bool ValueInList ( ListType  list , int  value , int  startIndex  )

/** Searches list for value between positions startIndex
               and  list.length-1
  @Pre  list.info[ startIndex ] . . list.info[ list.length - 1 ]
               contain values to be searched
  @Post  Function value = ( value exists in list.info[ startIndex ]
               . .  list.info[ list.length - 1 ] )   **/
{
   if  ( list.info[startIndex] == value )    // one base case
        return  true;
   else
   {
     if  (startIndex == list.length -1 ) // another base case
        return  false;
     else
        return ValueInList( list, value, startIndex + 1 );
   }
}
```

```
// ITERATIVE SOLUTION

bool ValueInList (  ListType  list , int value , int startIndex  )

/** Searches list for value between positions startIndex
               and  list.length-1
  @Pre  list.info[ startIndex ] . . list.info[ list.length - 1 ]
               contain values to be searched
  @Post Function value = ( value exists in list.info[ startIndex ]
               . .   list.info[ list.length - 1 ] )
{
   bool   found  =  false;
   while  ( !found  &&  startIndex < list.length )
   {
        if ( value == list.info[ startIndex ] )
           found = true;
        else
           startIndex++;
   }
   return  found;
}
```

Recursion and Efficiency

- Factors that contribute to inefficiency
  - Overhead associated with function calls
  - Some recursive algorithms inherently inefficient
    - repeated recursive calls with the same arguments
      - e.g. Fibonacci Sequence
- Keep in mind
  - Recursion can clarify complex solutions … but …
  - Clear, efficient iterative solutions are better

Use a recursive solution when:

- The depth of recursive calls is relatively “**shallow**” compared to the size of the problem
- The recursive version does **less** amount of work than the nonrecursive version
- The recursive version is [**much**] shorter and simpler than the nonrecursive solution

SHALLOW DEPTH

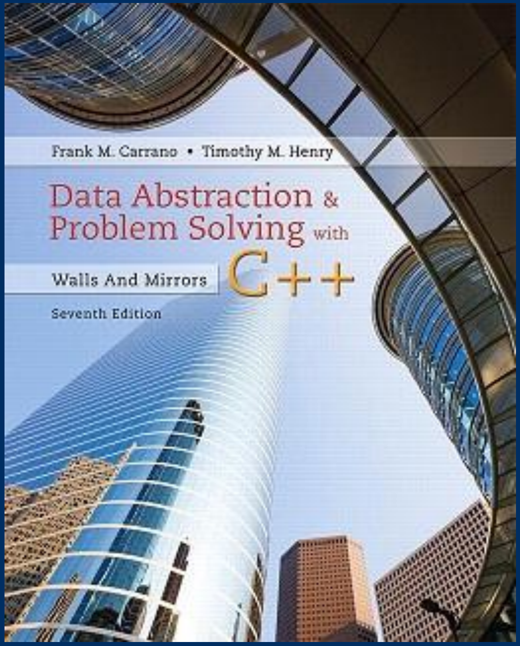
EFFICIENCY

CLARITY

The End

CS 302 - Data Structures

M. Abdullah Canbaz



Modified from authors' slides

Out of the box

- Binary Search

  https://www.geeksforgeeks.org/binary-search/

- Kth Smallest (or largest) Element

https://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array/