Chapter 2

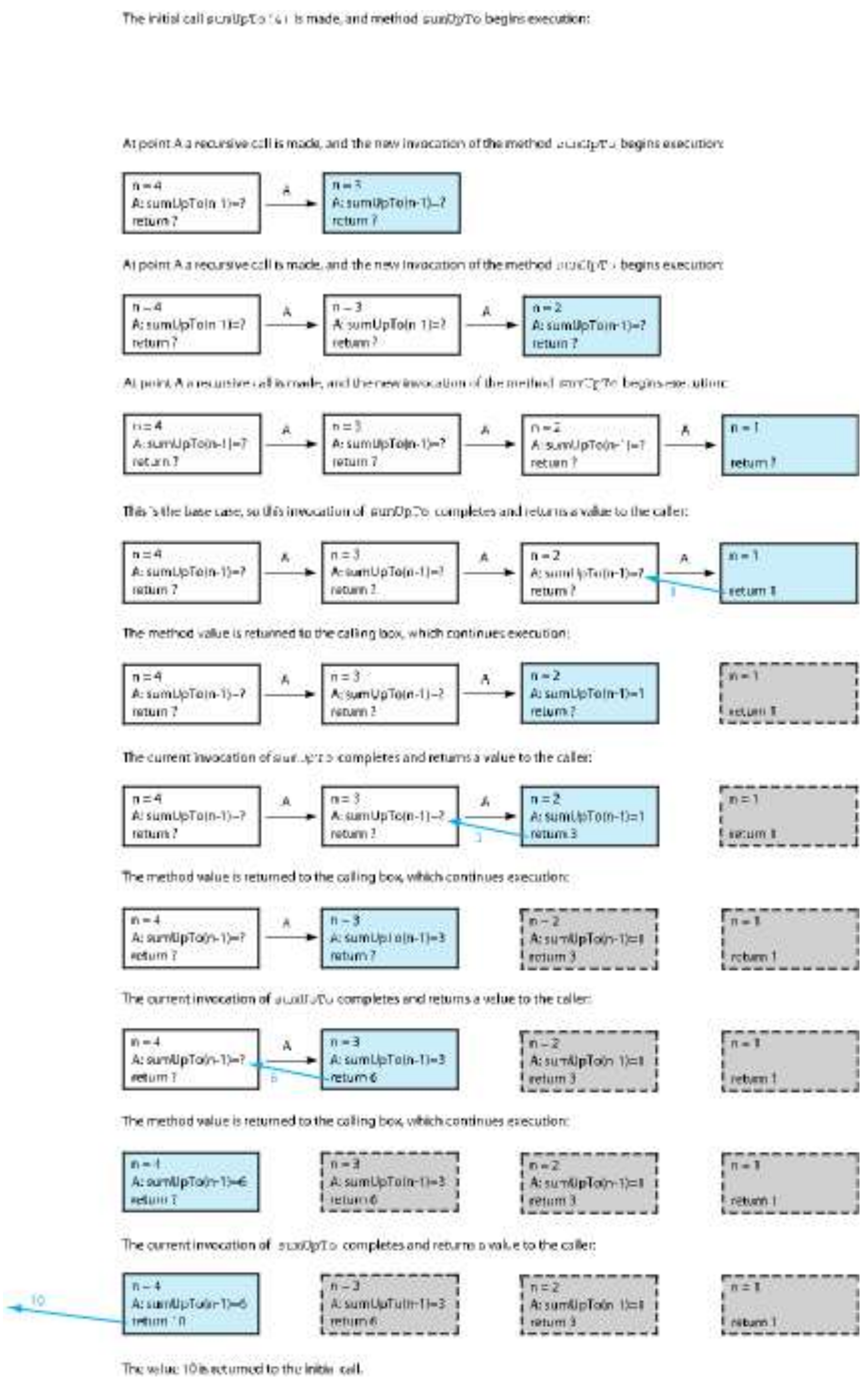
Question 1 The following function computes the sum of the first $n \geq 1$ integers. Show how this function satisfies the properties of a recursive function.

```
/** Computes the sum of the integers from 1 through n.
 @pre  n > 0.
 @post  None.
 @param n   A positive integer
 @return  The sum 1 + 2 + . . . + n. */
int sumUpTo(int n)
{
   int sum = 0;
   if (n == 1)
      sum = 1;
   else // n > 1
      sum = n + sumUpTo(n - 1);

   return sum;
}  // end sumUpTo
```

The product of n numbers is defined in terms of the product of $n - 1$ numbers, which is a smaller problem of the same type. When n is 1, the product is `anArray[0]`; this occurrence is the base case. Because $n \geq 1$ initially and n decreases by 1 at each recursive call, the base case will be reached.

Question 2 Write a box trace of the function given in Checkpoint Question 1.

We trace the function with 4 as its argument (see next page).



Question 3 Given an integer $n > 0$, write a recursive function `countDown` that writes the integers n, $n - 1$, . . . , 1. *Hint:* What task can you do and what task can you ask a friend to do for you?

```
// Precondition: n > 0.
// Postcondition: Writes n, n - 1, ... , 1.
void countDown(int n)
{
   if (n > 0)
   {
      cout << n << endl;
      countDown(n-1);
```

```
   }  // end if
}  // end countDown
```

Question 4 In the previous definition of `writeArrayBackward`, why does the base case occur when the value of `first` exceeds the value of `last`?

When `first` > `last`, the array is empty. That is the base case. Since the body of the `if` statement is skipped in this case, no action takes place.

Question 5 Write a recursive function that computes and returns the product of the first $n \geq 1$ real numbers in an array.

```
// Precondition: anArray is an array of n real numbers, n ≥ 1.
// Postcondition: Returns the product of the n numbers in
// anArray.
double computeProduct(const double anArray[], int
n), {
   if (n == 1)
      return anArray[0];
   else
      return anArray[n - 1] * computeProduct(anArray, n - 1);
}  // end computeProduct
```

Question 6 Show how the function that you wrote for the previous question satisfies the properties of a recursive function.

1. `computeProduct` calls itself.
2. An array of `n` numbers is passed to the method. The recursive call is given a smaller array of `n - 1` numbers.
3. `anArray[0]` is the base case.
4. Since `n ≥ 1` and the number of entries considered in `anArray` decreases by 1 at each recursive call, eventually the recursive call is `computeProduct(anArray, 1)`. That is, `n` is 1, and the base case is reached.

Question 7 Write a recursive function that computes and returns the product of the integers in the array `anArray[first..last]`.

```
// Precondition: anArray[first..last] is an array of integers,
// where first <= last.
// Postcondition: Returns the product of the integers in
// anArray[first..last].
double computeProduct(const int anArray[], int first, int last)
{
   if (first == last)
      return anArray[first];
   else
      return anArray[last] * computeProduct(anArray, first, last - 1);
}  // end computeProduct
```

Question 8 Define the recursive C++ function `maxArray` that returns the largest value in an array and adheres to the pseudocode just given.

```
// Precondition: anArray[first..last] is an array of integers,
// where first <= last.
// Postcondition: Returns the largest integer in
// anArray[first..last].
double maxArray(const int anArray[], int first, int last)
```

```
{
   if (first == last)
      return anArray[first];
   else
   {
      int mid = first + (last - first) / 2;
      return max(maxArray(anArray, first, mid),
                 maxArray(anArray, mid + 1, last))
   }  // end if
}  // end maxArray
```

Question 9 Trace the execution of the function `solveTowers` to solve the Towers of Hanoi problem for two disks.

The three recursive calls result in the following moves: Move a disk from *A* to *C*, from *A* to *B*, and then from *C* to *B*.

Question 10 Compute $g(4, 2)$.

6.

Question 11 Of the following recursive functions that you saw in this chapter, identify those that exhibit tail recursion: `fact`, `writeBackward`, `writeBackward2`, `rabbit`, *P* in the parade problem, `getNumberOfGroups`, `maxArray`, `binarySearch`, and `kSmall`.

`writeBackward`, `binarySearch`, and `kSmall`.

Chapter 2 Recursion: The Mirrors

1

- The problem is defined in terms of a smaller problem of the same type:

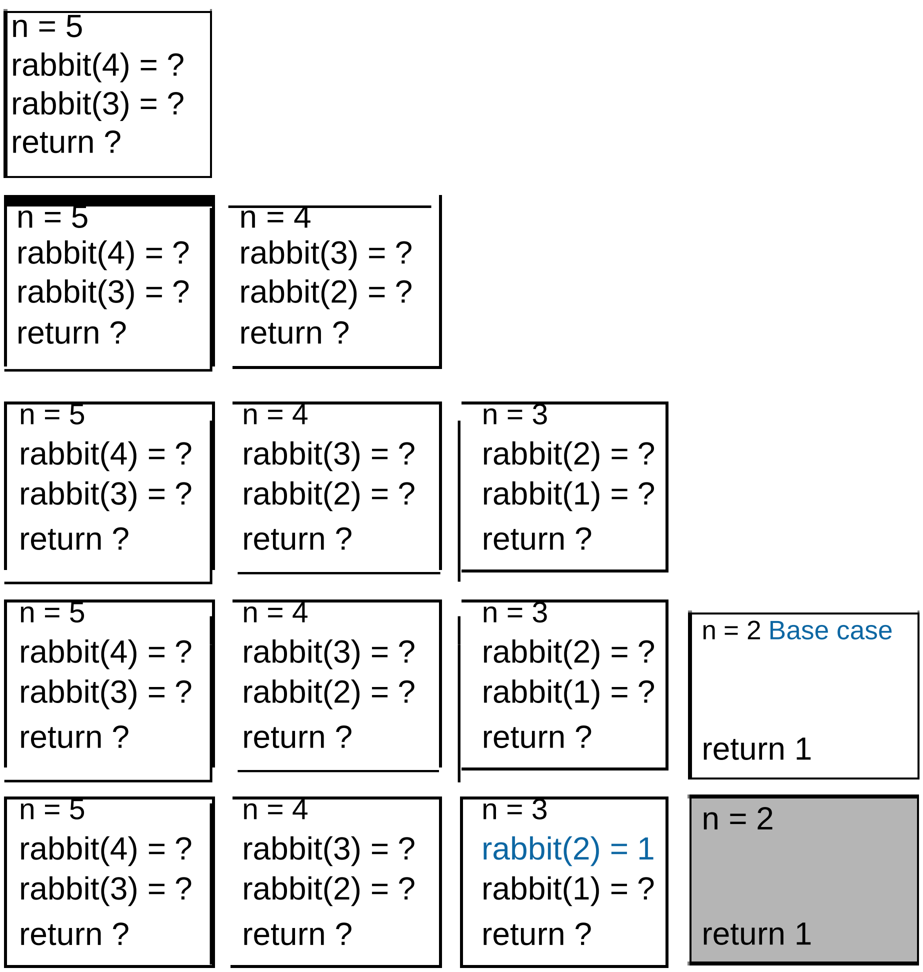
 Here, the last value in the array is checked and then the remaining part of the array is passed to the function.

- Each recursive call diminishes the size of the problem: The recursive call to `getNumberEqual` subtracts 1 from the current value of `n` and passes this value as the argument `n` in the next call, effectively reducing the size of the unsearched remainder of the array by 1.
- An instance of the problem serves as the base case: When the size of the array is 0 (i.e.: n δ 0), the function returns 0; that is, an array of size 0 can have no occurrences of `desiredValue`. This case terminates the recursion.
- As the problem size diminishes, the base case is reached: `n` is an integer and is decremented by 1 with each recursive call.

The argument `n` in the *n*th recursive call will have the value 0, and the base case will be reached.

2a

The call `rabbit(5)` produces the following box trace:



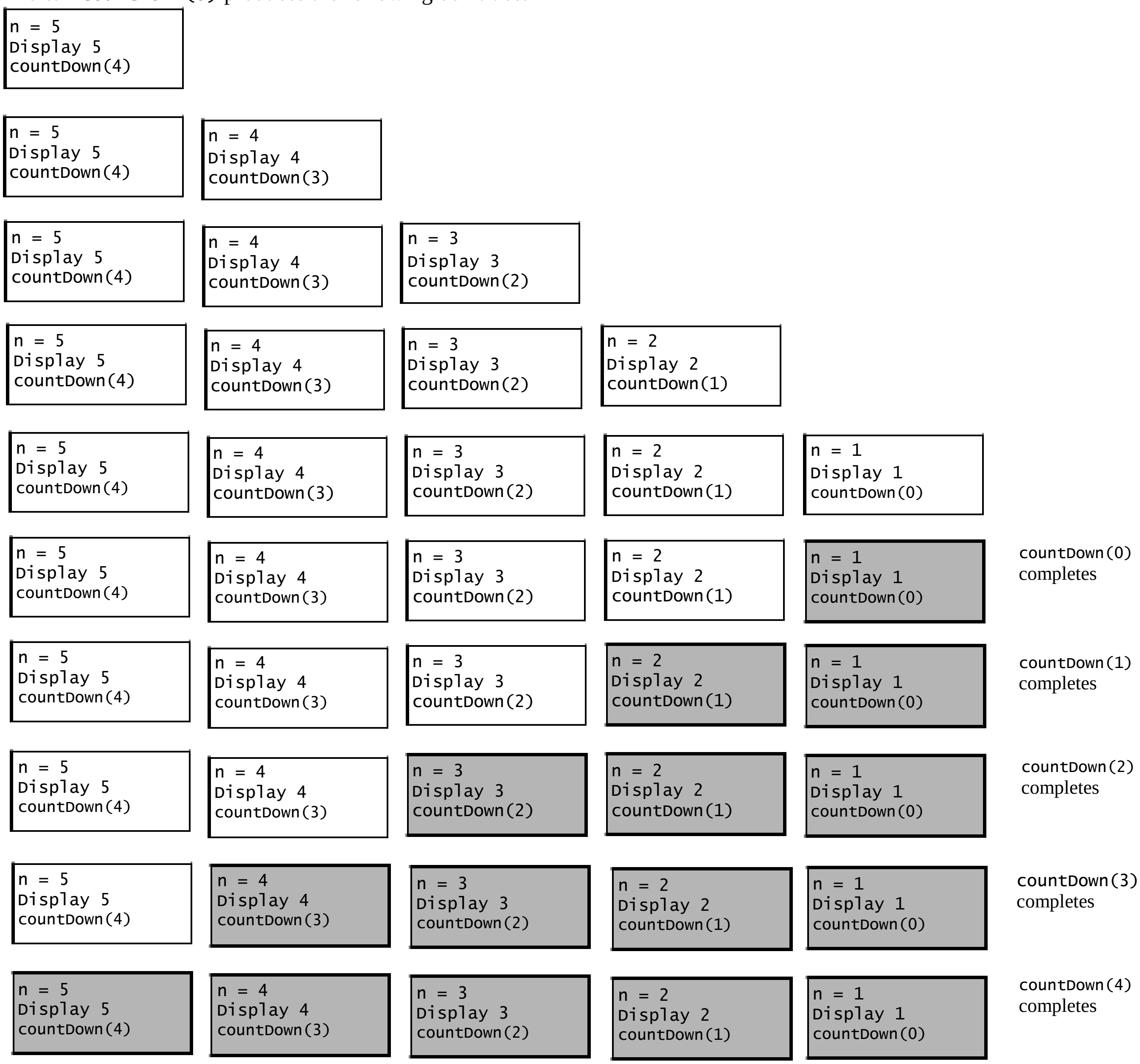
n = 5 rabbit(4) = ? rabbit(3) = ? return ?	n = 4 rabbit(3) = ? rabbit(2) = ? return ?	n = 3 rabbit(2) = 1 rabbit(1) = ? return ?	n = 1 Base case return 1
n = 5 rabbit(4) = ? rabbit(3) = ? return ?	n = 4 rabbit(3) = ? rabbit(2) = ? return ?	n = 3 rabbit(2) = 1 rabbit(1) = 1 return 2	n = 1 return 1
n = 5 rabbit(4) = ? rabbit(3) = ? return ?	n = 4 rabbit(3) = 2 rabbit(2) = ? return ?	n = 3 rabbit(2) = 1 rabbit(1) = 1 return 2	n = 1 return 1
n = 5 rabbit(4) = ? rabbit(3) = ? return ?	n = 4 rabbit(3) = 2 rabbit(2) = ? return ?	n = 2 Base case return 1	
n = 5 rabbit(4) = ? rabbit(3) = ? return ?	n = 4 rabbit(3) = 2 rabbit(2) = 1 return 3	n = 2 return 1	
n = 5 rabbit(4) = 3 rabbit(3) = ? return ?	n = 4 rabbit(3) = 2 rabbit(2) = 1 return 3	n = 2 return 1	
n = 5 rabbit(4) = 3 rabbit(3) = ? return ?	n = 3 rabbit(2) = ? rabbit(1) = ? return ?		
n = 5 rabbit(4) = 3 rabbit(3) = ? return ?	n = 3 rabbit(2) = ? rabbit(1) = ? return ?	n = 2 Base case return 1	
n = 5 rabbit(4) = 3 rabbit(3) = ? return ?	n = 3 rabbit(2) = 1 rabbit(1) = ? return ?	n = 2 return 1	

n = 5 rabbit(4) = 3 rabbit(3) = ? return ?	n = 3 rabbit(2) = 1 rabbit(1) = ? return ?	n = 1 Base case return 1
n = 5 rabbit(4) = 3 rabbit(3) = ? return ?	n = 3 rabbit(2) = 1 rabbit(1) = 1 return 2	n = 1 return 1
n = 5 rabbit(4) = 3 rabbit(3) = 2 return 5	n = 3 rabbit(2) = 1 rabbit(1) = 1 return 2	n = 1 return 1
n = 5 rabbit(4) = 3 rabbit(3) = 2 return 5	n = 3 rabbit(2) = 1 rabbit(1) = 1 return 2	n = 1 return 1

The `rabbit(5)` call completes and the value 5 is returned to the calling function.

2b

The call `countDown(5)` produces the following box trace:



`countDown(5)` completes and returns to the calling function.

3

```cpp
/** Returns the sum of the first n integers in the array anArray.
 Precondition:  0 <= n <= size of anArray.
 Postcondition:  The sum of the first n integers in the array anArray is returned.
                 The contents of anArray and the value of n are unchanged. */
int computeSum(const int anArray[], int n)
{ // Base case
   if (n <= 0)
      return 0;
   else  // Reduce the problem size
      return anArray[n - 1] + computeSum(anArray, n - 1);
}  // end computeSum
```

4

```cpp
/** Returns the sum of the consecutive integers from start to end.
 Precondition:  start < end.
 Postcondition: The sum of the consecutive integers from start to end is returned.
                start and end are unchanged. */
int sum(int start, int end )
{
   if (start < end)
      return start + sum(start + 1, end);
   else
      return end;
}  // end sum
```

5a

```cpp
#include <string>
// Writes a character string backward.
// Precondition:  The string s is the string to write backward.
// Postcondition: s is written backward, but remains unchanged.
void writeBackward(std::string s)
{
   int length = s.size();
   if (length == 1)
      std::cout << s.substr(0, 1); // length == 1 is the base case
   else if (length > 1)
   {
      std::cout << s.substr(length – 1, 1);   // Write last character
      writeBackward(s.substr(0, length - 1)); // Write rest of string backward
   }  // end if
}  // end writeBackward
```

5b

```
#include <string>
// Writes a character string backward.
// Precondition:  The string s is the string to write backward.
// Postcondition: s is written backward, but remains unchanged.
void writeBackward2(std::string s)
{
   int length = s.size();
   if (length > 0)
   {
      // Write all but first character of string
      backward writeBackward2(s.substr(1, length - 1));
      // Write first character
      std::cout << s.substr(0, 1);
   }  // end if
   // length == 0 is the base case; do nothing
} // end writeBackward2
```

6

The recursive method does not have a base case. As such, it will never terminate.

7

```
/** Displays the integers from m through n.
 Precondition:  0 <= m <= n.
 Postcondition: The integers from m through n are displayed on one line. */
void writeIntegers(int m, int n)
{
   std::cout << m << " ";
   if (m < n)
   {
      writeIntegers(m + 1, n);
   }  // end if
}  // end writeIntegers
```

8

```
/** Returns the sum of the squares of 1 to n.
 Precondition:  n > 0.
 Postcondition: sum of the squares of 1 to n is returned. */
int sumOfSquares(int n)
{
   int result;
   if (n == 1)
      result = 1;
   else
      result = n * n + sumOfSquares(n - 1);
   return result;
}  // end sumOfSquares
```

9

```
const int NUMBER_BASE = 10;

/** Displays the decimal digits of an integer in reverse order.
 Precondition:  integer >= 0.
 Postcondition: The decimal digits of integer are displayed in reverse order.
 This function does not output a newline character at the end of a string. */
void reverseDigits(int integer)
{
   if (integer >= 0)
   {  // Base case
      if (integer < NUMBER_BASE)
         std::cout << integer;
      else
      { // Display rightmost digit std::cout
         << integer % NUMBER_BASE;

         // Display remaining digits in reverse order
         reverseDigits(integer / NUMBER_BASE);
      } // end if
   } // end if
}  // end reverseDigits
```

10a

```
/** Displays a line of n characters, where ch is the character.
 Precondition:   n >= 0.
 Postcondition: A line of n characters ch is output
    followed by a newline. */
void writeLine(char ch, int n)
{ // Base case
   if (n <= 0)
      std::cout << std::endl;

   // Write rest of
   line else
   {
      std::cout << ch;

      writeLine(ch, n - 1);
   }  // end if
}  // end writeLine
```

10b

```
/** Displays a block of m rows of n occurrences of the character ch.
 Precondition:  m >= 0 and n >= 0.
 Postcondition:  A block of m rows by n columns of character ch is displayed. */
void writeBlock(char ch, int m, int n)
{
   if (m > 0)
   {
      writeLine(ch, n);          // Write first line
      writeBlock(ch, m - 1, n); // Write rest of
   block } // end if
   // Base case: m <= 0 do nothing.
} // end writeBlock
```

11

```
Enter: a = 1 b = 7
Enter: a = 1 b = 3
Leave: a = 1 b = 3
Leave: a = 1 b = 7
2
```

12

```
mystery(30) produces the following output:
Enter: first = 1 last = 30
Enter: first = 1 last = 14
Enter: first = 1 last = 6
Enter: first = 4 last = 6
Leave: first = 4 last = 6
Leave: first = 1 last = 6
Leave: first = 1 last = 14
Leave: first = 1 last = 30
mystery(30) = 5; should be 5
```

13

The given function first checks to see whether *n* is a positive number. If not, it immediately terminates. Otherwise, an integer division of *n* by 8 is taken, and if the result is greater than 0 (i.e.: if $n > 8$), the function is called again with *n*/8 as an argument. This call processes that portion of the number composed of higher powers of 8. After this call, the residue for the current power, *n* % 8, is printed.

The function computes the number of times 8^0, 8^1, 8^2, ... will divide *n*. These values are stacked recursively and are displayed in the reverse of the order of computation. The following is the hand execution with $n = 100$:

```
displayOctal(100)
   displayOctal(12)
      displayOctal(1)
      Display 1 % 8, or 1
   Display 12 % 8, or 4
Display 100 % 8, or 4
```

The final output is 144.

14

```
The value of f(8) is
Function entered with n = 8
Function entered with n = 6
Function entered with n = 4
Function entered with n = 2
Function entered with n = 0
Function entered with n = 2
Function entered with n = 4
Function entered with n = 2
Function entered with n = 0
27
```

Even though the precondition for the function `f` states that its argument `n` is nonnegative, no actual code in `f` prevents a negative value for `n`. For `n` larger than 2, the value of `f(n)` is the sum of `f(n-2)` and `f(n-4)`. If `n` is even, `n-2` and `n-4` are the next two smaller even integers; likewise, if `n` is odd, `n-2` and `n-4` are the next two smaller odd integers. Thus any odd nonnegative integer `n` will eventually cause `f(n)` to evaluate `f(3)`. Because 3 is not within the range of 0 to 2, the `switch` statement's default case will execute, and the function will recursively call `f(1)` and `f(-1)`. Once `n` becomes negative, the recursive calls that `f(n)` makes will never reach a base case. Theoretically, we will have an infinite sequence of function calls, but in practice an exception will occur.

15

The following output is produced when `x` is a value argument:

```
6 2
7 1
8 0
8 0
7 1
6 2
```

Changing `x` to a reference argument produces:

```
6 2
7 1
8 0
8 0
8 1
8 2
```

16a

The box trace for the call `binSearch(a, 0, 7, 5)` follows:

target = 5
first = 0
last = 7
mid = 3
target < a[3]
index = binSearch(a,0,2,5)
return ?

target = 5
first = 0
last = 2
mid = 1
target == a[1]
index = 1 Base case
return 1

target = 5
first = 0
last = 7
mid = 3
target < a[3]
index = 1
return 1

target = 5
first = 0
last = 2
mid = 1
target == a[1]
index = 1
return 1

16b

The box trace for the call `binSearch(a, 0, 7, 13)` follows:

target = 13
first = 0
last = 7
mid = 3
target > a[3]
index = binSearch(a,4,7,13)
return ?

target = 13
first = 4
last = 7
mid = 5
target < a[5]
index = binSearch(a,4,4,13)
return ?

target = 13
first = 4
last = 4
mid = 4
target < a[4]
index = binSearch(a,4,3,13)
return ?

target = 13
first = 4 last =
3 first > last

index = -1 Base case
return -1

target = 13
first = 0
last = 7
mid = 3
target > a[3]
index = binSearch(a,4,7,13)
return ?

target = 13
first = 4
last = 7
mid = 5
target < a[5]
index = binSearch(a,4,4,13)
return ?

target = 13
first = 4 last
= 4 mid = 4

target < a[4]
index = -1
return -1

target = 13
first = 4
last = 3
first > last
index = -1
return -1

target = 13
first = 0
last = 7
mid = 3
target > a[3]
index = binSearch(a,4,7,13)
return ?

target = 13
first = 4
last = 7
mid = 5
target < a[5]
index = -1
return -1

target = 13
first = 4 last
= 4 mid = 4

target < a[4]
index = -1
return -1

target = 13
first = 4
last = 3
first > last
index = -1
return -1

target = 13
first = 0
last = 7
mid = 3
target > a[3]

target = 13
first = 4
last = 7
mid = 5
target < a[5]

target = 13 first
= 4 last = 4 mid
= 4 target < a[4]

target = 13
first = 4
last = 3
first > last
index = -1

16c

The box trace for the call `binSearch(a, 0, 7, 16)` follows:

target = 16 first = 0 last = 7 mid = 3 target > a[3] index = `binSearch(a,4,7,16)` return ?	target = 16 first = 4 last = 7 mid = 5 target < a[5] index = `binSearch(a,4,4,16)` return ?	target = 16 first = 4 last = 4 mid = 4 target > a[4] index = `binSearch(a,5,4,16)` return ?	target = 16 first = 4 last = 3 first > last index = -1 `Base case` return -1
target = 16 first = 0 last = 7 mid = 3 target > a[3] index = `binSearch(a,4,7,16)` return ?	target = 16 first = 4 last = 7 mid = 5 target < a[5] index = `binSearch(a,4,4,16)` return ?	target = 16 first = 4 last = 4 mid = 4 target > a[4] `index = -1` return -1	target = 13 first = 4 last = 3 first > last index = -1 return -1
target = 16 first = 0 last = 7 mid = 3 target > a[3] index = `binSearch(a,4,7,16)` return ?	target = 16 first = 4 last = 7 mid = 5 target < a[5] `index = -1` return -1	target = 16 first = 4 last = 4 mid = 4 target > a[4] `index = -1` return -1	target = 16 first = 4 last = 3 first > last index = -1 return -1
target = 16 first = 0 last = 7 mid = 3 target > a[3] `index = -1` return -1	target = 16 first = 4 last = 7 mid = 5 target < a[5] `index = -1` return -1	target = 16 first = 4 last = 4 mid = 4 target > a[4] `index = -1` return -1	target = 16 first = 4 last = 3 first > last index = -1 return -1

18

a. For a binary search to work, the array must first be sorted in either ascending or descending order.

b. The index is (0 + 102) / 2 = 50.

c. Number of comparisons = $\lfloor \log 101 \rfloor = 6$.

19

```
/** Returns the value of x raised to the nth power.
    Precondition:  n >= 0
    Postcondition: The computed value is returned. */
double power1(double x, int n) {

   double result = 1; // Value of x^0

   while (n > 0)     // Iterate until n == 0
   { result *= x;
      n--;
   } // end while
   return result;
}  // end power1

/** Returns the value of x raised to the nth power.
    Precondition:  n >= 0
    Postcondition: The computed value is returned. */
double power2(double x, int n) {

   if (n == 0)
      return 1; // Base case
   else
      return x * power2(x, n-1);
}  // end power2

/** Returns the value of x raised to the xth power.
    Precondition:  n >= 0
    Postcondition: The computed value is returned. */
double power3(double x, int n) {

   if (n == 0)
      return 1;
   else
   {
      double halfPower = power3(x, n/2);

      // if n is even...
      if (n % 2 == 0)
         return halfPower * halfPower;
      else // if n is odd...
         return x * halfPower *
   halfPower; } // end if
}  // end power3
```

19d

	3^{32}	3^{19}
power1	32	19
power2	32	19
power3	7	8

19e

	3^{32}	3^{19}
power2	32	19
power3	6	5

20

Maintain a count of the recursive depth of each call by passing this count as an additional argument to the rabbit function; indent that many spaces or tabs in front of each line of output.

```
/** Computes a term in the Fibonacci sequence.
 Precondition:  n is a positive integer and tab > 0.
 Postcondition: The progress of the recursive function call is displayed
    as a sequence of increasingly nested blocks. The function
    returns the nth Fibonacci number. */
int rabbit(int n, int tab)
{
   int value;

   // Indent the proper distance for this
   block for (int i = 0; i < tab; i++)
      std::cout << " ";

   // Display status of call
   std::cout << "Enter rabbit: n = " << n << std::endl;

    if (n <= 2)
       value = 1;
    else  // n  > 2, so n-1 > 0 and n-2 > 0;
          // indent by one for next call
       value =  rabbit(n - 1, 2 * tab) + rabbit(n - 2, 2 * tab);

   // Indent the proper distance for this
   block for (int i = 0; i < tab; i++)
      std::cout << " ";

   // Display status of call
   std::cout << "Leave rabbit: n = " << n << " value = " << value << std::endl;

   return value;
}  // end rabbit
```

21a

```
// Recursive version. Pre: n > 0.
int fOfNforPartA(int n)
{
   int result;
   switch(n)
   {
      case 1: case 2: case
         3: result = 1;
         break;
      case 4: result
         = 3; break;

      case 5: result
         = 5; break;

      default: // n > 5
         result = fOfNforPartA(n - 1) + 3 * fOfNforPartA(n -
         5); break;
   }  // end switch

   return result;
}  // end fOfNforPartA
```

f(6) is 8; *f*(7) is 11; *f*(12) is 95; *f*(15) is 320.

21b

Since we only need the five most recently computed values, we will maintain a "circular" five-element array indexed modulus 5.

```
// Iterative version. Pre: n > 0.
int fOfNforPartB(int n)
{
   int last5[5] = {1, 1, 1, 3, 5}; // Values of f(1) through
   f(5) int result;

   if (n < 6)
      result = last5[n - 1];
   else  // n >= 6
   {
      for (int i = 5; i < n; i++)
      {
         result = last5[(i - 1) % 5] + 3 * last5[(i - 5) % 5];

         // Replace entry in last5
         last5[i % 5] = result;  // f(i) = f(i - 1) + 3 x f(i - 5)
      }  // end for

      result = last5[(n - 1) % 5];
   }  // end if

   return result;
}  // end fOfNforPartB
```

22

```
// Computes n! iteratively. n >= 0.
long fact(int n)
{
   long result = 1.0;

   if (n > 1)
   {
      for (int i = 2; i <= n; i++)
         result *= i;
   }  // end if

   return result;
}  // end fact

// Writes a string backwards iteratively.
void writeBackward(std::string str)
{
   for (int i = str.size() - 1; i >= 0; i--
      ) std::cout << str[i];

   std::cout << std::endl;
}  // end writeBackward

/** Iteratively searches a sorted array; returns either the index of the array element
    containing a value equal to the given target or -1 if no such element exists. */
int binarySearch(int anArray[], int target, int first, int last)
{
   int result = -1;
   while (first < last)
   {
      int mid = first + (last - first) / 2;

      if (anArray[mid] == target)
      {
         first = mid;
         last = mid;
      }
      else if (anArray[mid] < target)
         first = mid + 1; // Search the upper half
      else
         last = mid - 1;  // Search the lower half
   }  // end while

   if (first > last)
      result = -1; // If not found, return -1
   elseif (anArray[first] != target)
      result = -1;
   else
      result = first;

   return result;
}  // end binarySearch
```

23

Discovering the loop invariant will easier if we first convert the `for` loop to a `while` loop:

```
int previous = 1; // Initially rabbit(1)
int current = 1; // Initially rabbit(2)
int next = 1; // rabbit(n); initial value when n is 1 or 2
// Compute next rabbit values when n >= 3
int i = 3;
while (i <= n)
{
   // current is rabbit(i – 1), previous is rabbit(i –
   2) next = current + previous; // rabbit(i)
   previous = current; // Get ready for next
   iteration current = next;
   i++;
}  // end while
```

Before the loop: $i = 3$, current = rabbit(i – 1) = rabbit(2), and previous = rabbit(i – 2) = rabbit(1).
At the beginning of the loop's body: $3 \le i \le n$, current = rabbit(i – 1), and previous = rabbit(i – 2).
At the end of the loop's body: $4 \le i \le n + 1$, next = rabbit(i – 1), current = rabbit(i – 1), and previous = rabbit(i – 2).
After the loop ends, next = rabbit(n).

24a

Prove: If a and b are positive integers with $a > b$ such that b is not a divisor of a, then $gcd(a, b) = gcd(b, a \bmod b)$.

Let $d = gcd(a, b)$. Then, $a = dj$ and $b = dk$ for integers d , j and k. Now let $n = a \bmod b$. Then $(n - a)/b = q$, where q is an integer. So, $n - a = bq$, or $n - dj = dkq$. That is, $n = d(kq + j)$. Then, $(n/d) = kq + j$, where $(kq + j)$ is an integer. So, d divides n; That is, d divides (a mod b).

To show that d is the greatest common divisor of b and a mod b, assume that it is not. That is, assume there exists an integer $g > d$ such that $b = gr$ and $(a \bmod b) = gs$ for integers r and s. Then, $(gs - a)/gr = q'$ where q' is an integer. So $gs - a = grq'$. Thus, $a = g(s - rq')$. We have that g divides a, and g divides b. But $gcd(a, b) = d$. This contradiction indicates that our assumption was incorrect. Therefore, $gcd(b, a \bmod b) = d = gcd(a, b) = d$.

24b

If $b > a$, $a \bmod b = a$. Therefore, $gcd(a, b) = gcd(b, a \bmod b) = gcd(b, a)$. The arguments a and $b$are reversed.

24c

When $a > b$, the argument associated with the parameter a in the next recursive call is b, which is smaller than a. If $b > a$, the next recursive call will swap the arguments so that $a > b$. Thus, the first argument will eventually equal the second and so eventually a mod b will be 0. That is, the base case will be reached.

25a

$$c(n) = \begin{cases} 0 & \text{if} \quad n = 1 \\ 1 & \text{if} \quad n = 2 \\ \sum_{i=1}^{n-1} (c(n-i) + 1) & \text{if} \quad n > 2 \end{cases}$$

25b

$$c(n) = \begin{cases} 0 & \text{if} \quad n = 1 \\ 1 & \text{if} \quad n = 2 \\ c(n-1) + c(n-2) & \text{if} \quad n > 2 \end{cases}$$

26

Acker(1, 2) = 4.

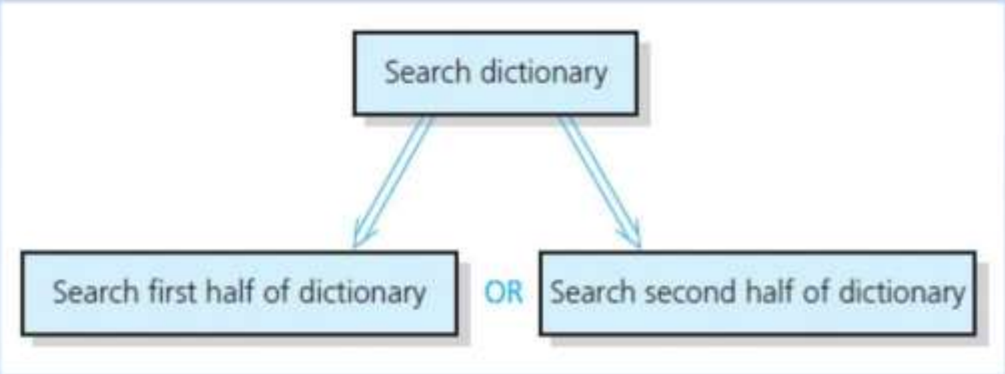
```
int acker(int m, int n)
{
   int result;

   if (m ==  0)
      result = n + 1;
   else if (n == 0)
      result = acker(m - 1, 1);
   else
      result = acker(m - 1, acker(m, n - 1));

   return result;
}  // end acker
```

Recursion: The Mirrors

Chapter 2

Recursive Solutions

- Recursion breaks problem into smaller identical problems
  - An alternative to iteration
- FIGURE 2-1 A recursive solution



Recursive Solutions

- A recursive function calls itself
- Each recursive call solves an identical, but smaller, problem
- Test for base case enables recursive calls to stop
- Eventually, one of smaller problems must be the 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

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

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

A Recursive Valued Function: The Factorial of *n*

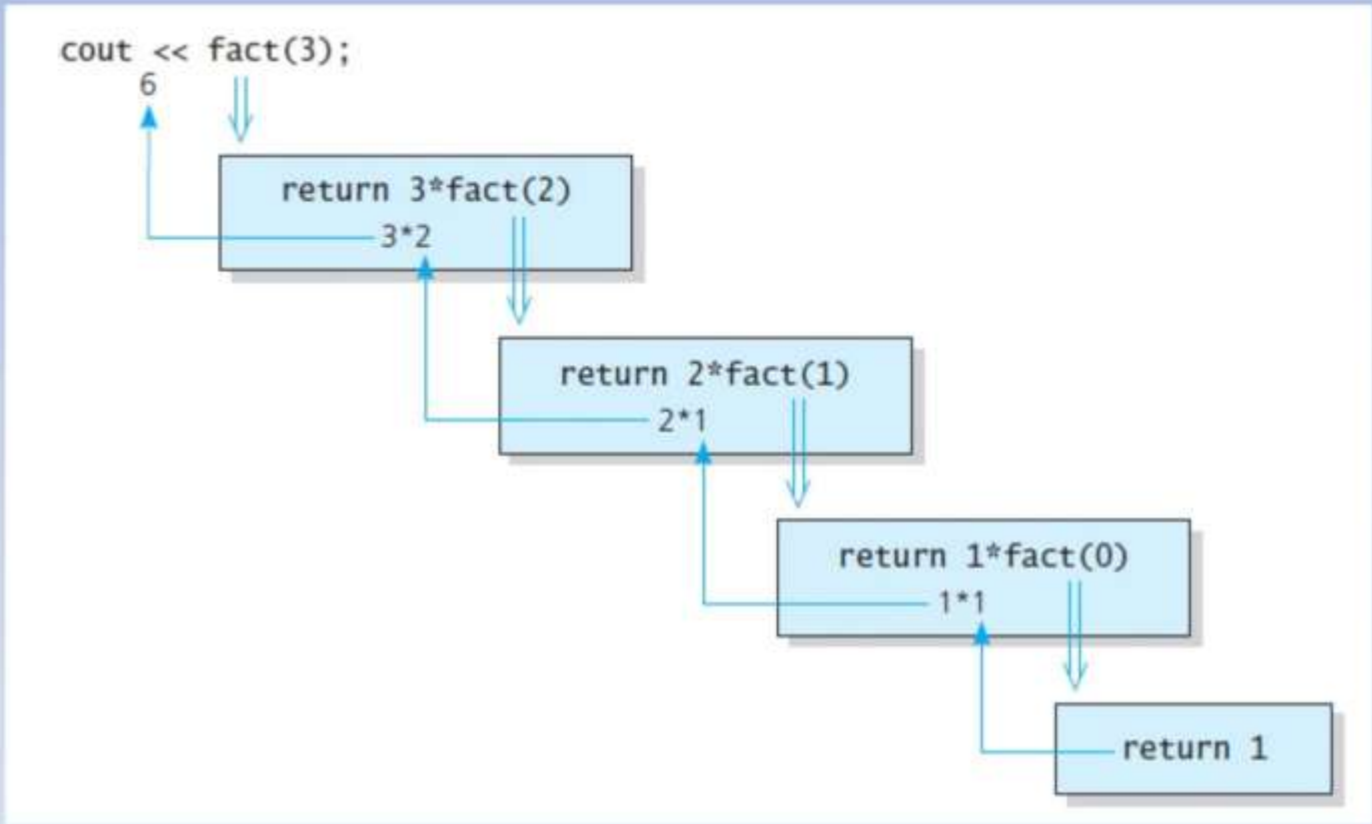


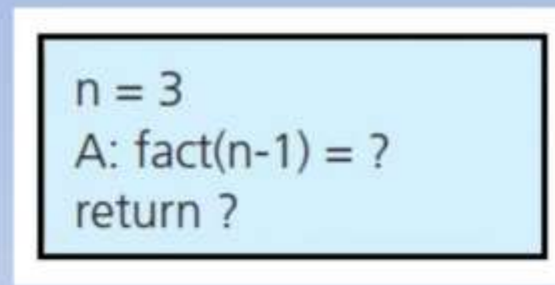
FIGURE 2-2 fact(3)

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

FIGURE 2-3 A box



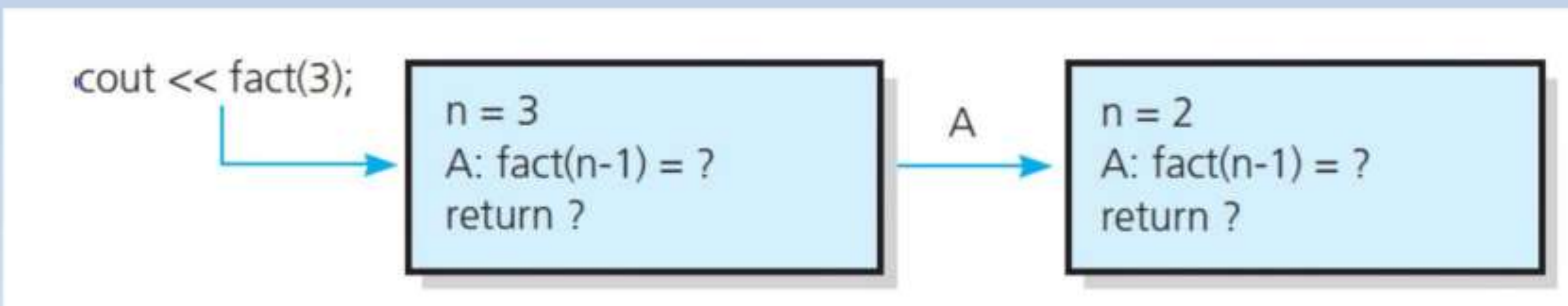


FIGURE 2-4 The beginning of the box trace

The Box Trace

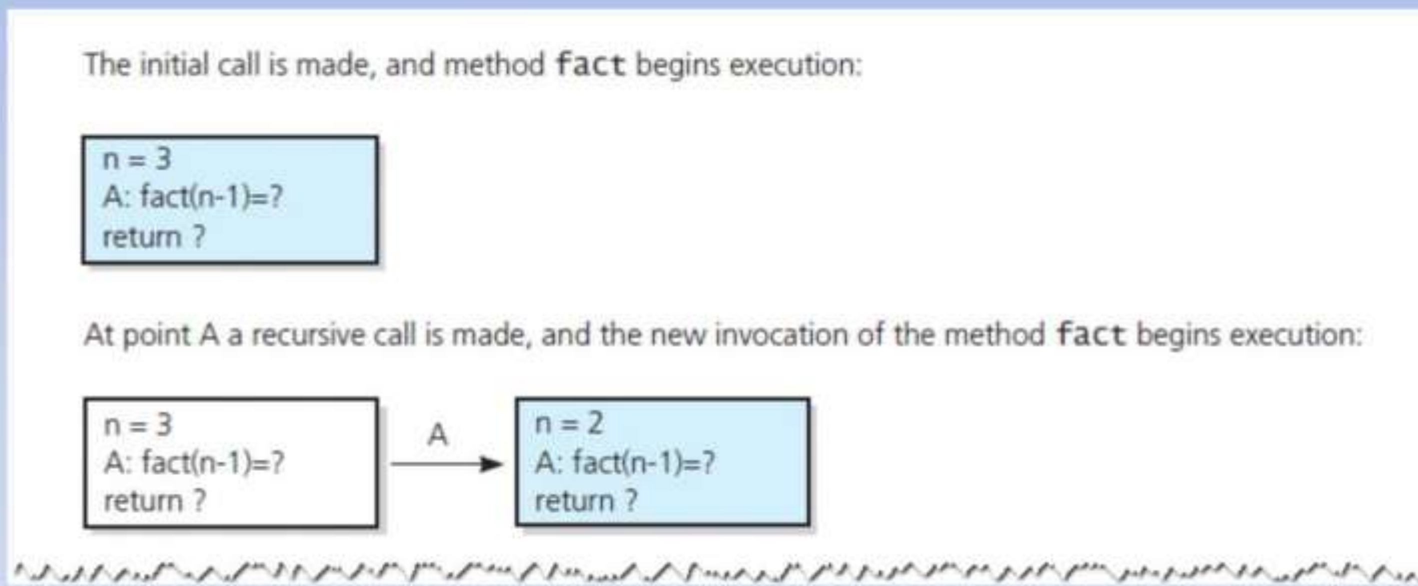


FIGURE 2-5 Box trace of fact(3)

The Box Trace

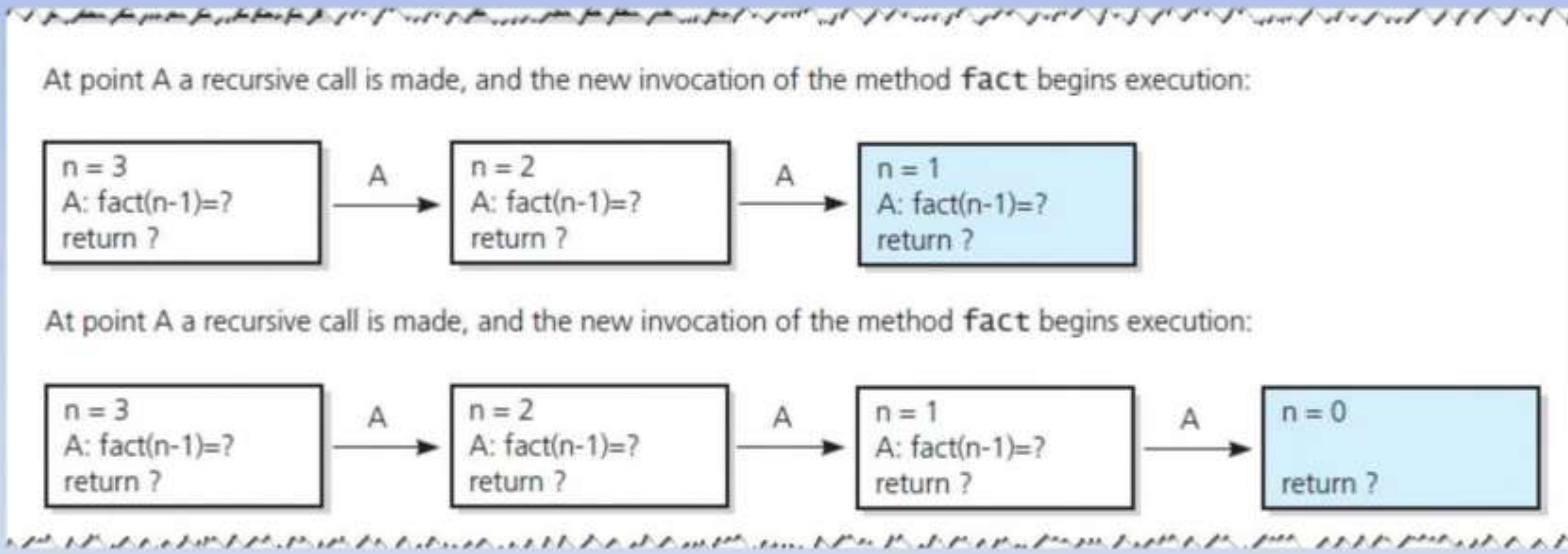


FIGURE 2-5 Box trace of fact(3)

The Box Trace

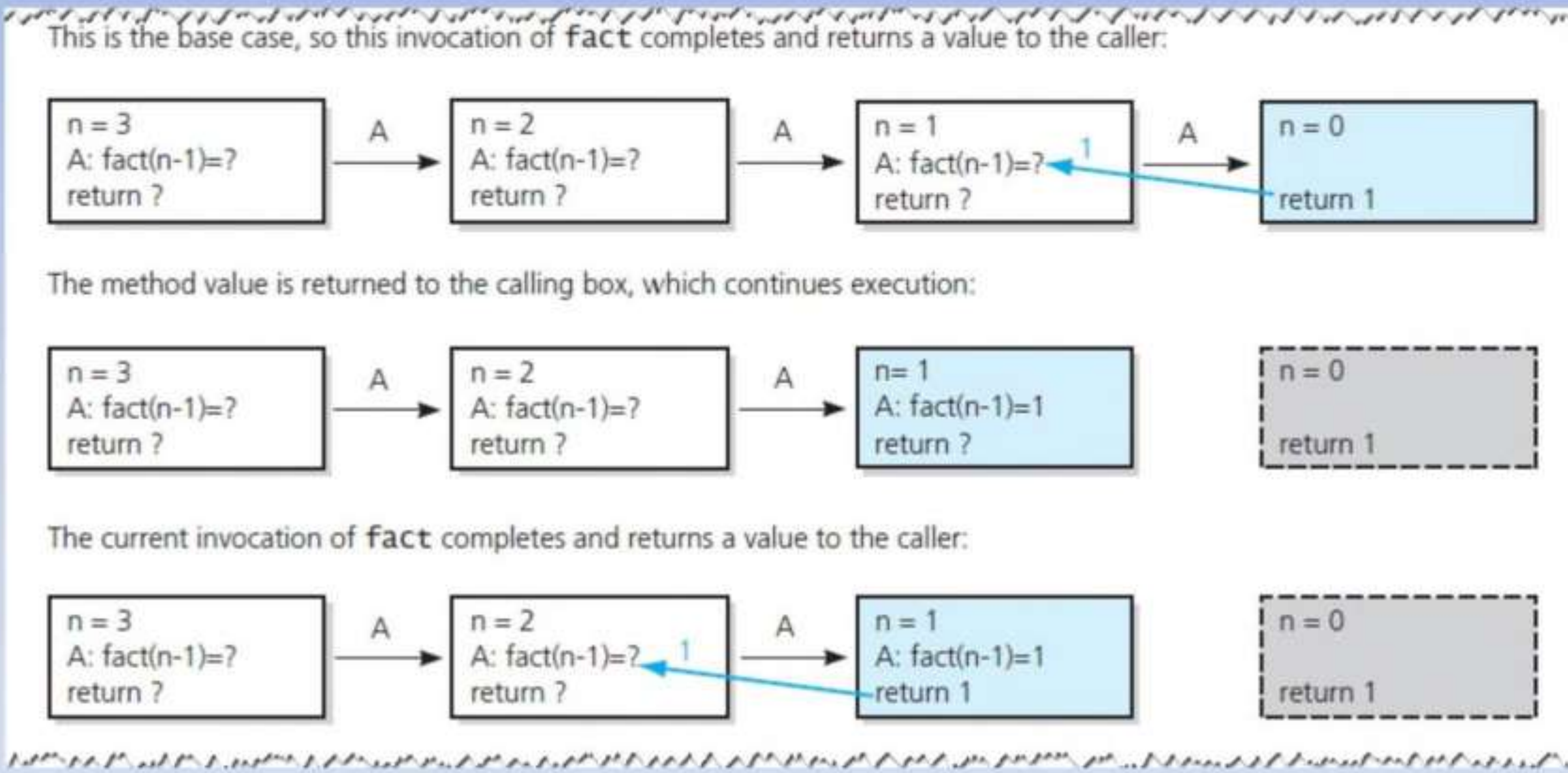


FIGURE 2-5 Box trace of fact(3)

The Box Trace

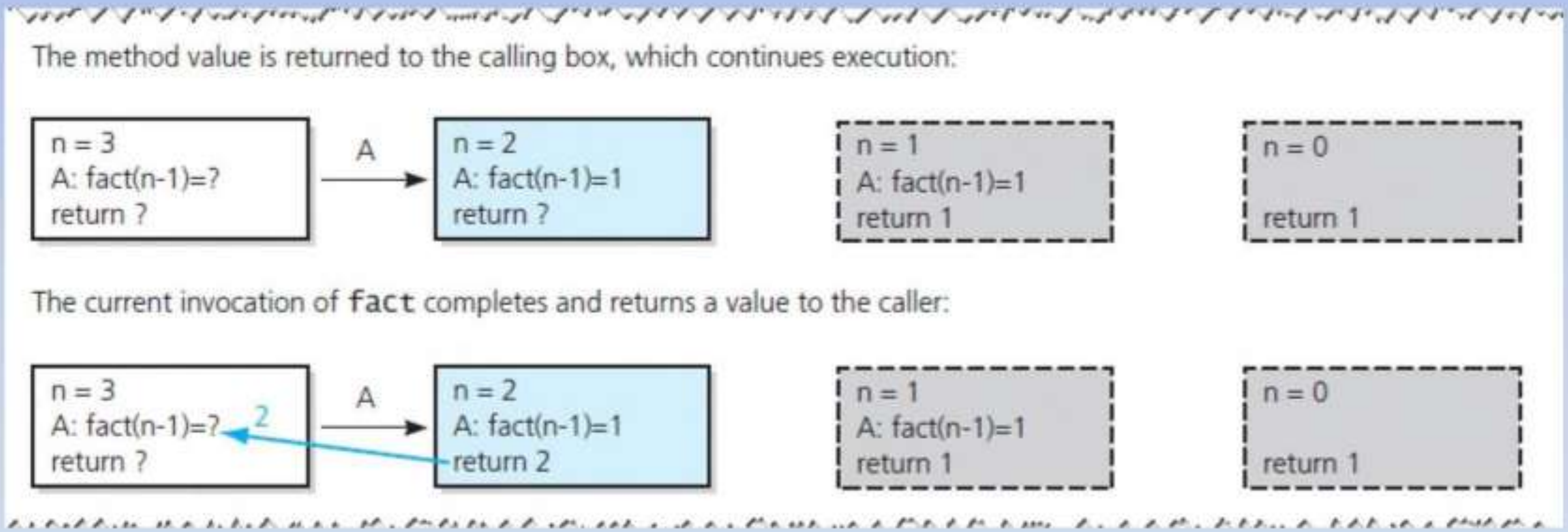


FIGURE 2-5 Box trace of fact(3)

The Box Trace

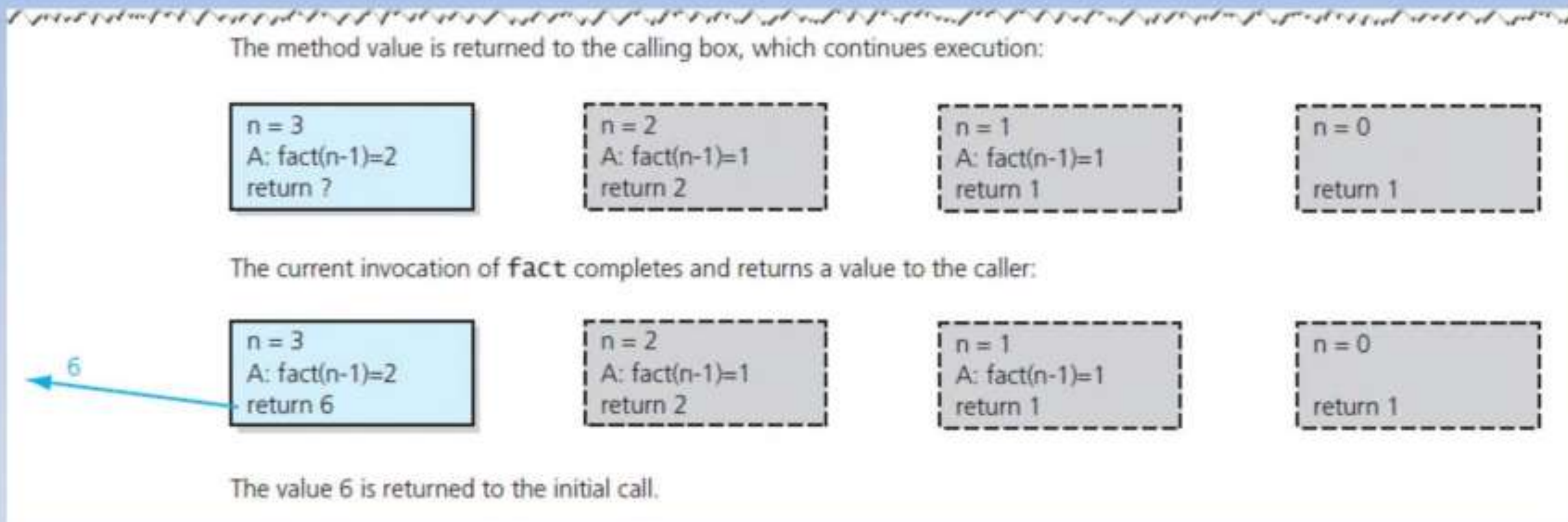


FIGURE 2-5 Box trace of fact(3)

A Recursive Void Function: Writing a String Backward

- Likely candidate for minor task is writing a single character.
  - Possible solution: strip away the last character

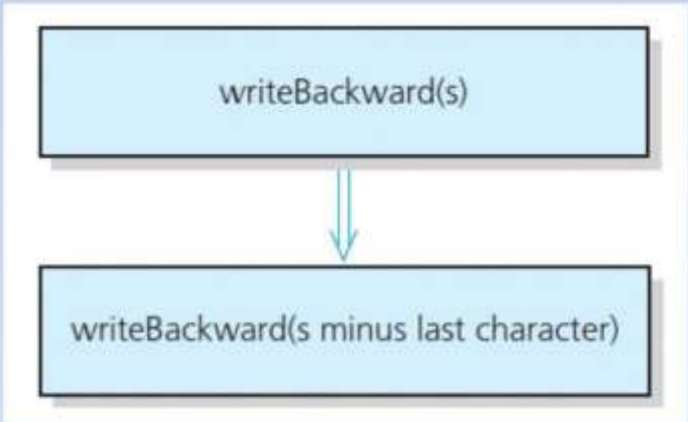


FIGURE 2-6 A recursive solution

A Recursive Void Function: Writing a String Backward

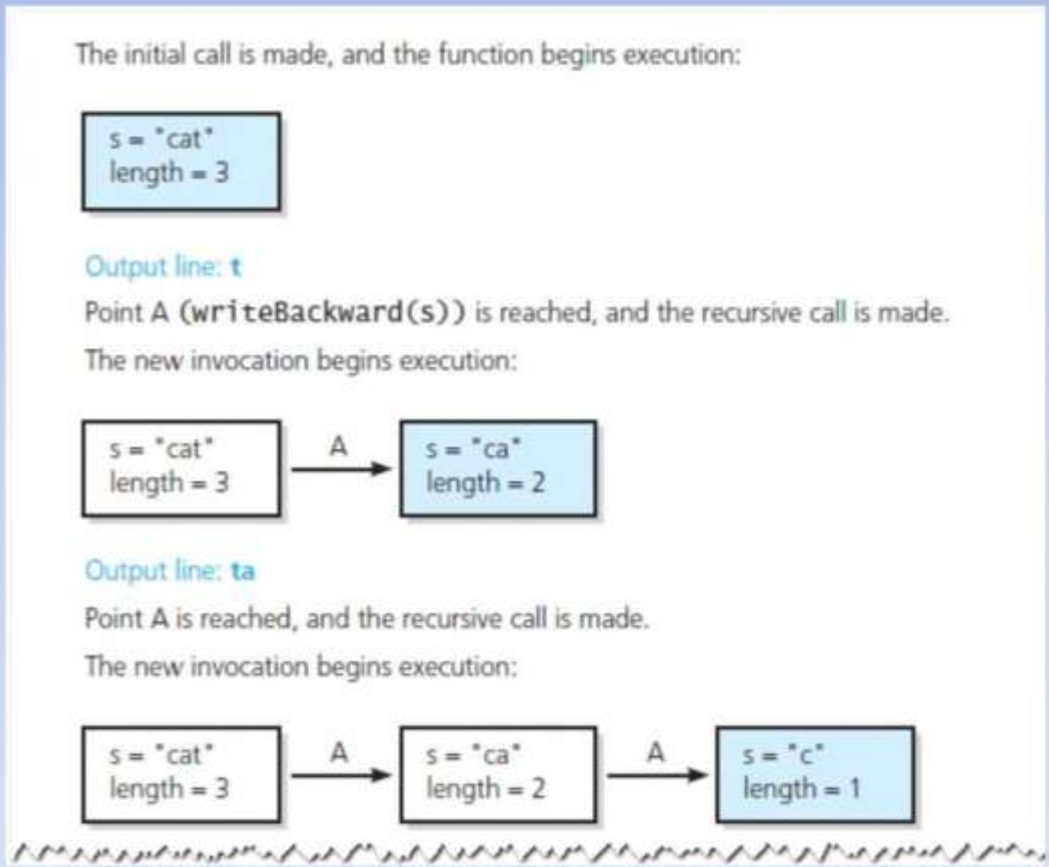


FIGURE 2-7 Box trace of writeBackward("cat")

A Recursive Void Function: Writing a String Backward

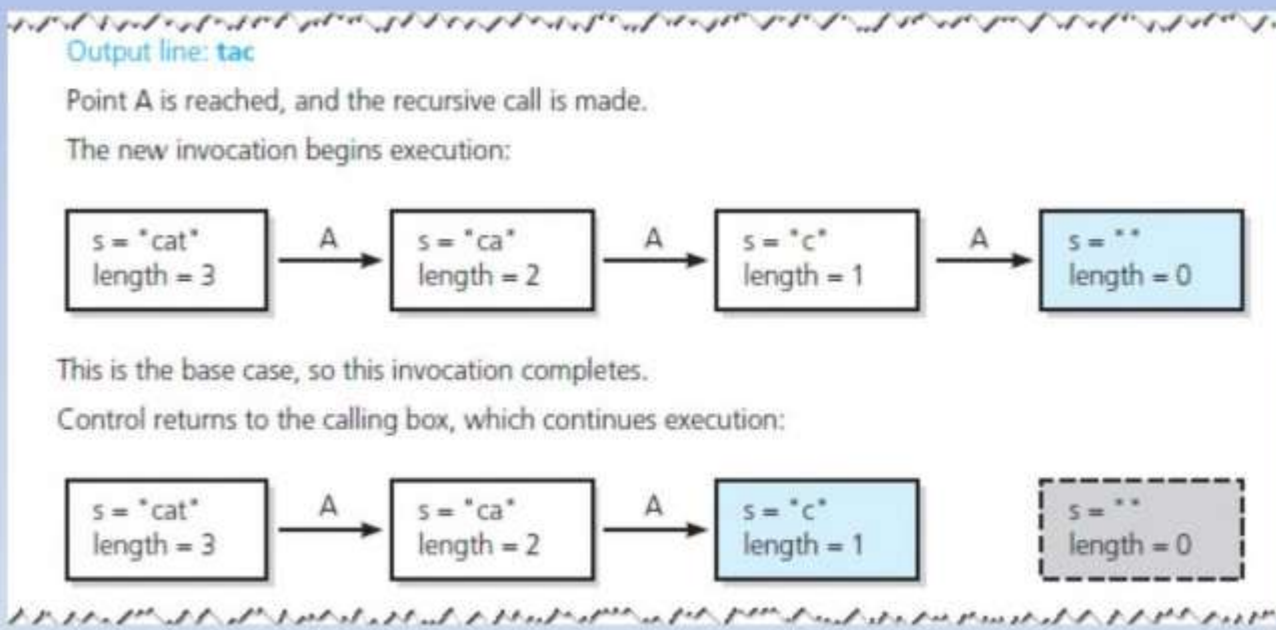


FIGURE 2-7 Box trace of writeBackward("cat")

A Recursive Void Function: Writing a String Backward

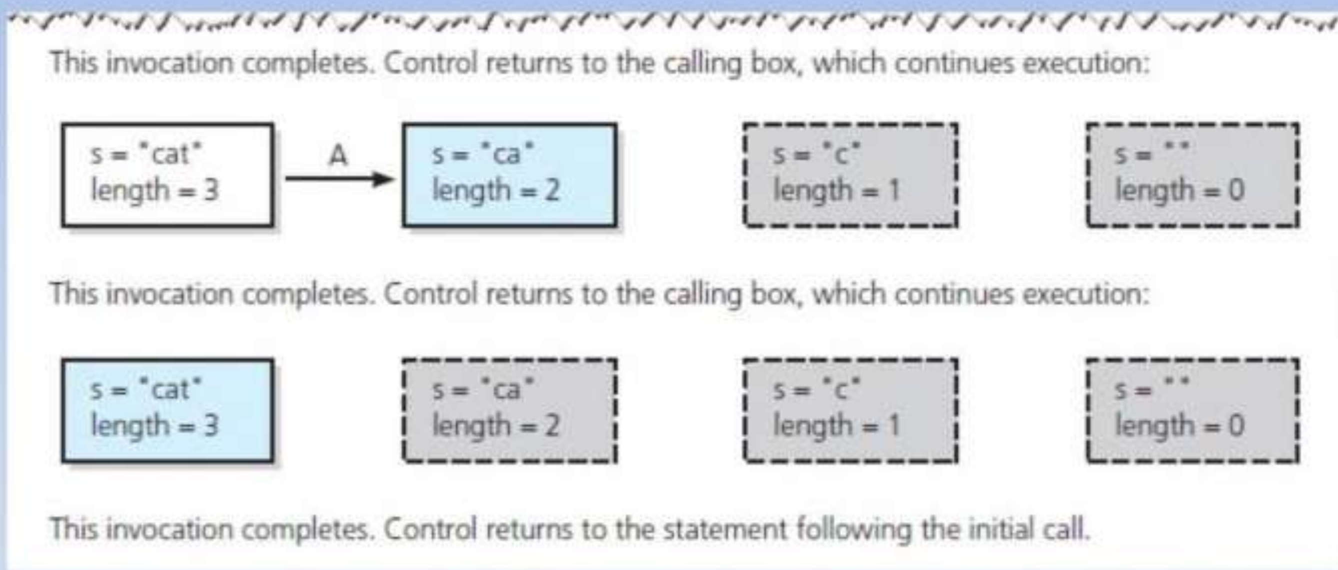


FIGURE 2-7 Box trace of writeBackward("cat")

A Recursive Void Function: Writing a String Backward

- Another possible solution
  - Strip away the first character

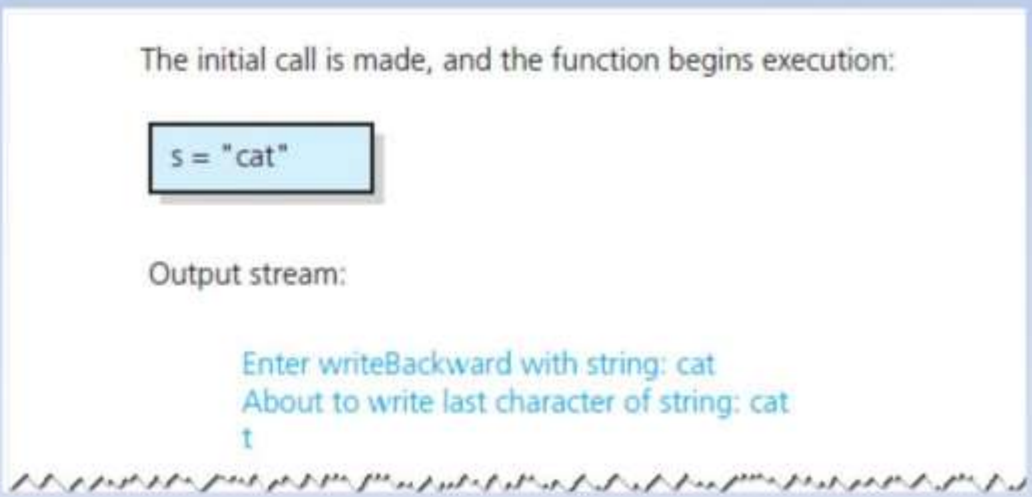


FIGURE 2-8 Box trace of writeBackward("cat") in pseudocode

A Recursive Void Function: Writing a String Backward

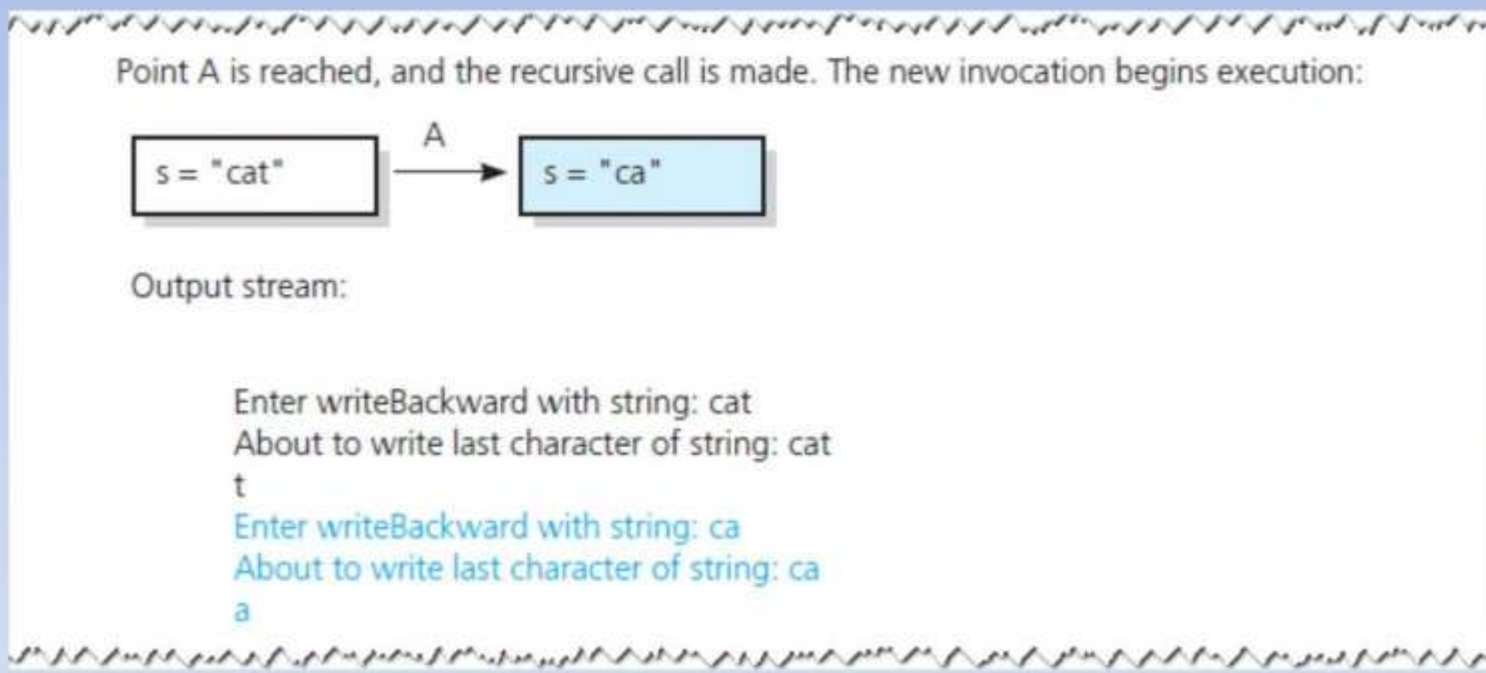


FIGURE 2-8 Box trace of writeBackward("cat") in pseudocode

A Recursive Void Function: Writing a String Backward

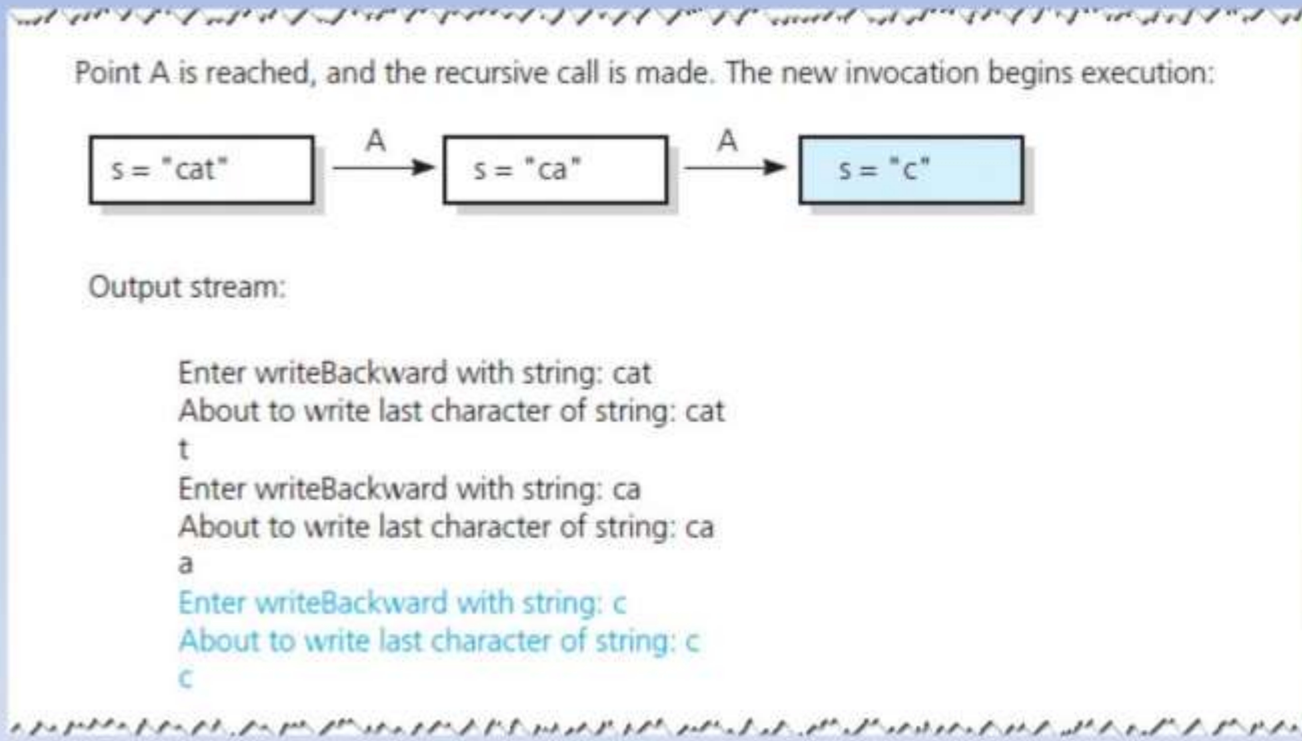


FIGURE 2-8 Box trace of writeBackward("cat") in pseudocode

A Recursive Void Function: Writing a String Backward

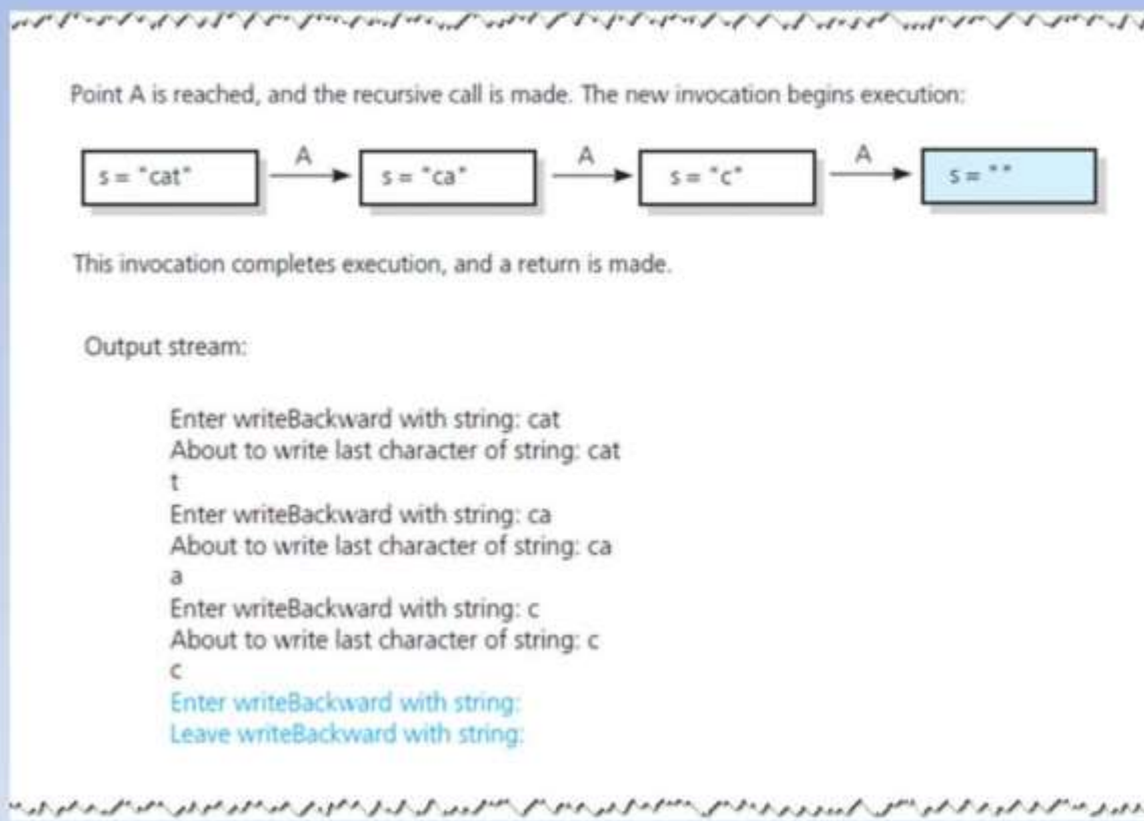
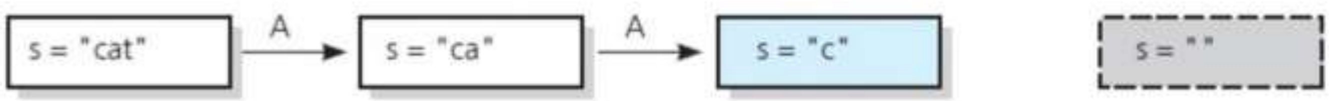


FIGURE 2-8 Box trace of writeBackward("cat") in pseudocode

A Recursive Void Function: Writing a String Backward



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
c
Enter writeBackward with string:
Leave writeBackward with string:
Leave writeBackward with string: c
```

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

A Recursive Void Function: Writing a String Backward

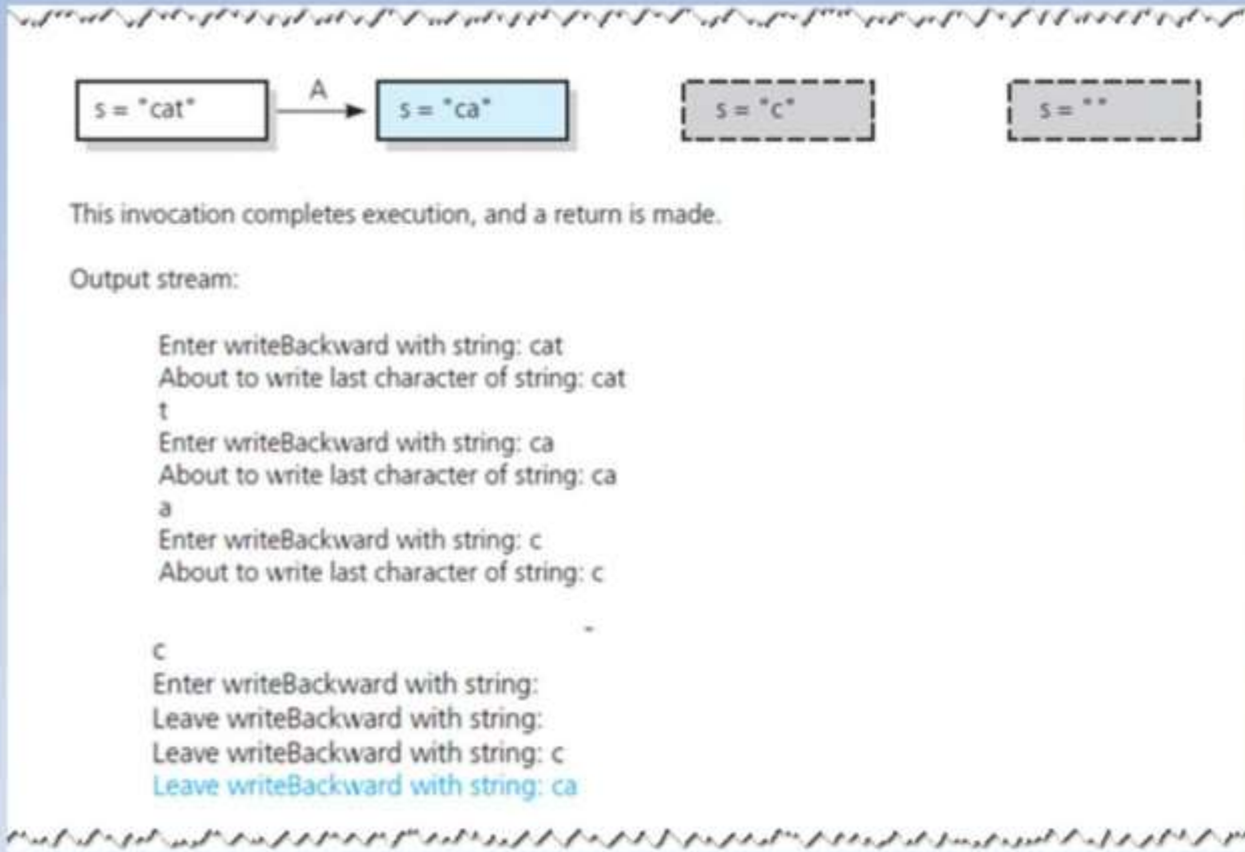


FIGURE 2-8 Box trace of writeBackward("cat") in pseudocode

A Recursive Void Function: Writing a String Backward

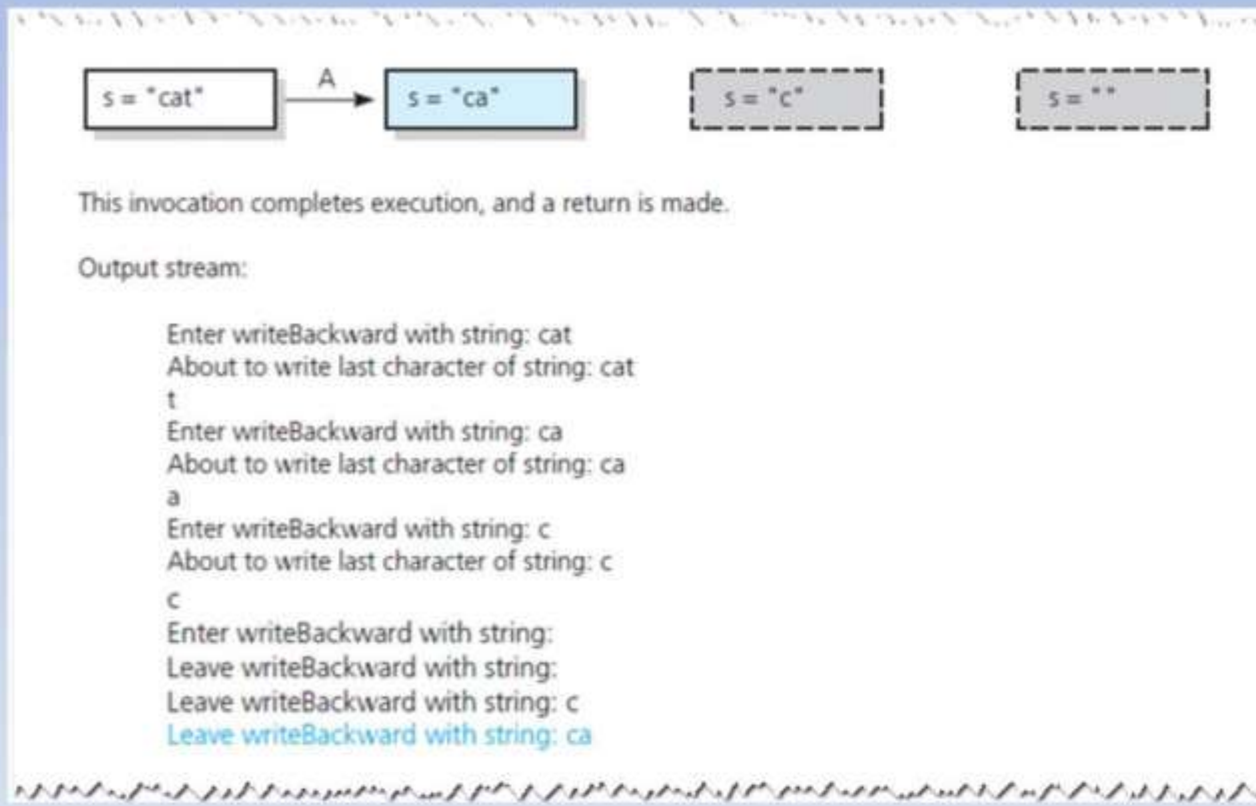


FIGURE 2-8 Box trace of writeBackward("cat") in pseudocode

A Recursive Void Function: Writing a String Backward

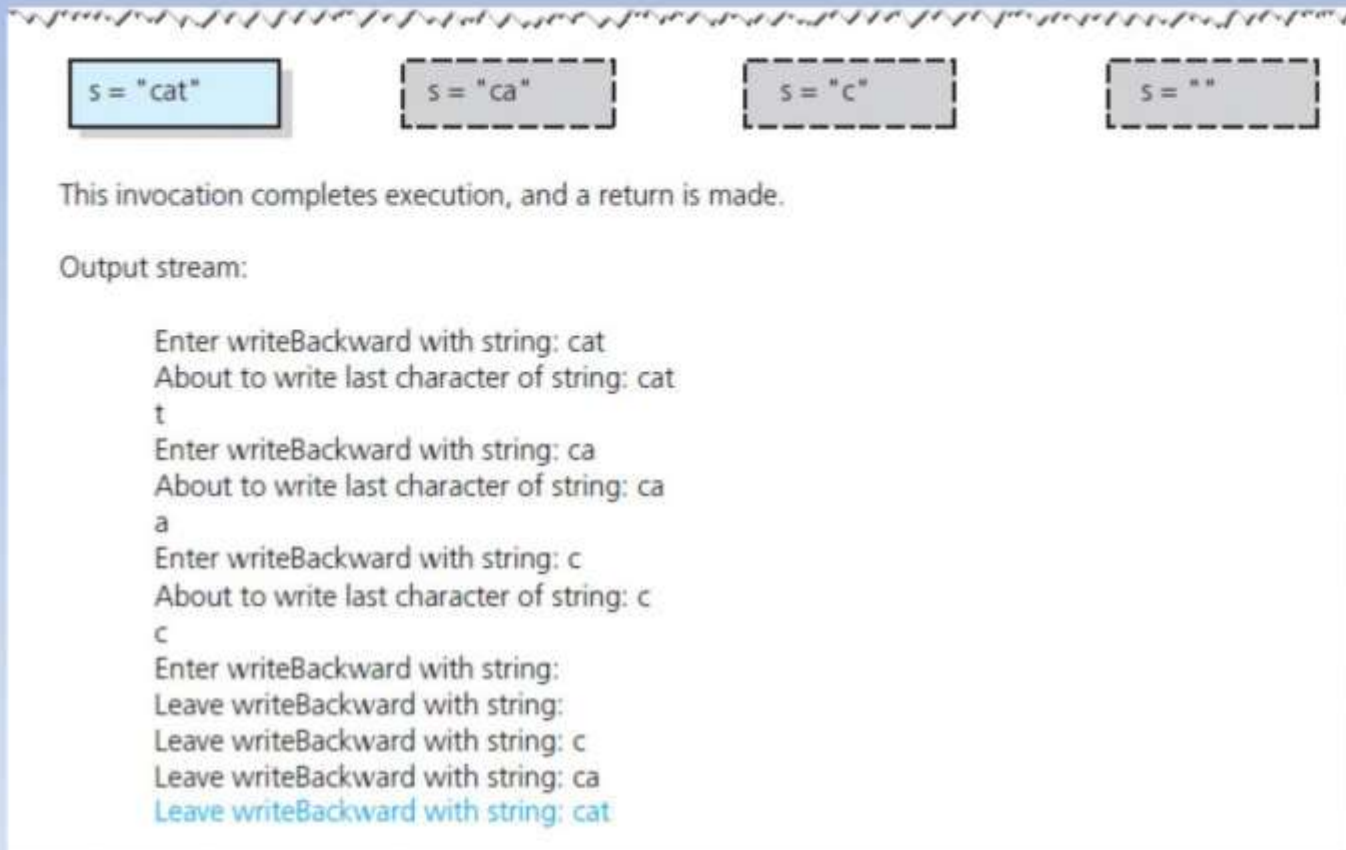


FIGURE 2-8 Box trace of writeBackward("cat") in pseudocode

A Recursive Void Function: Writing a String Backward

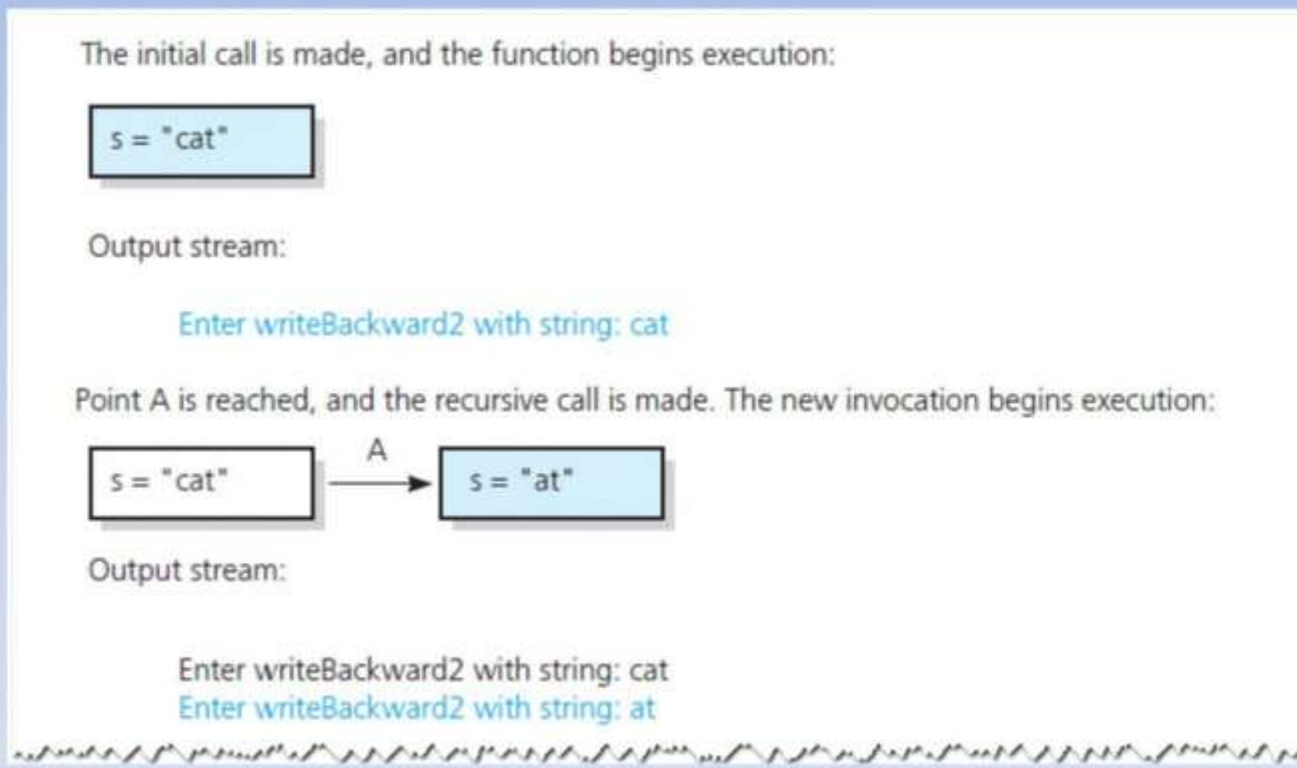


FIGURE 2-8 Box trace of writeBackward2("cat") in pseudocode

A Recursive Void Function: Writing a String Backward

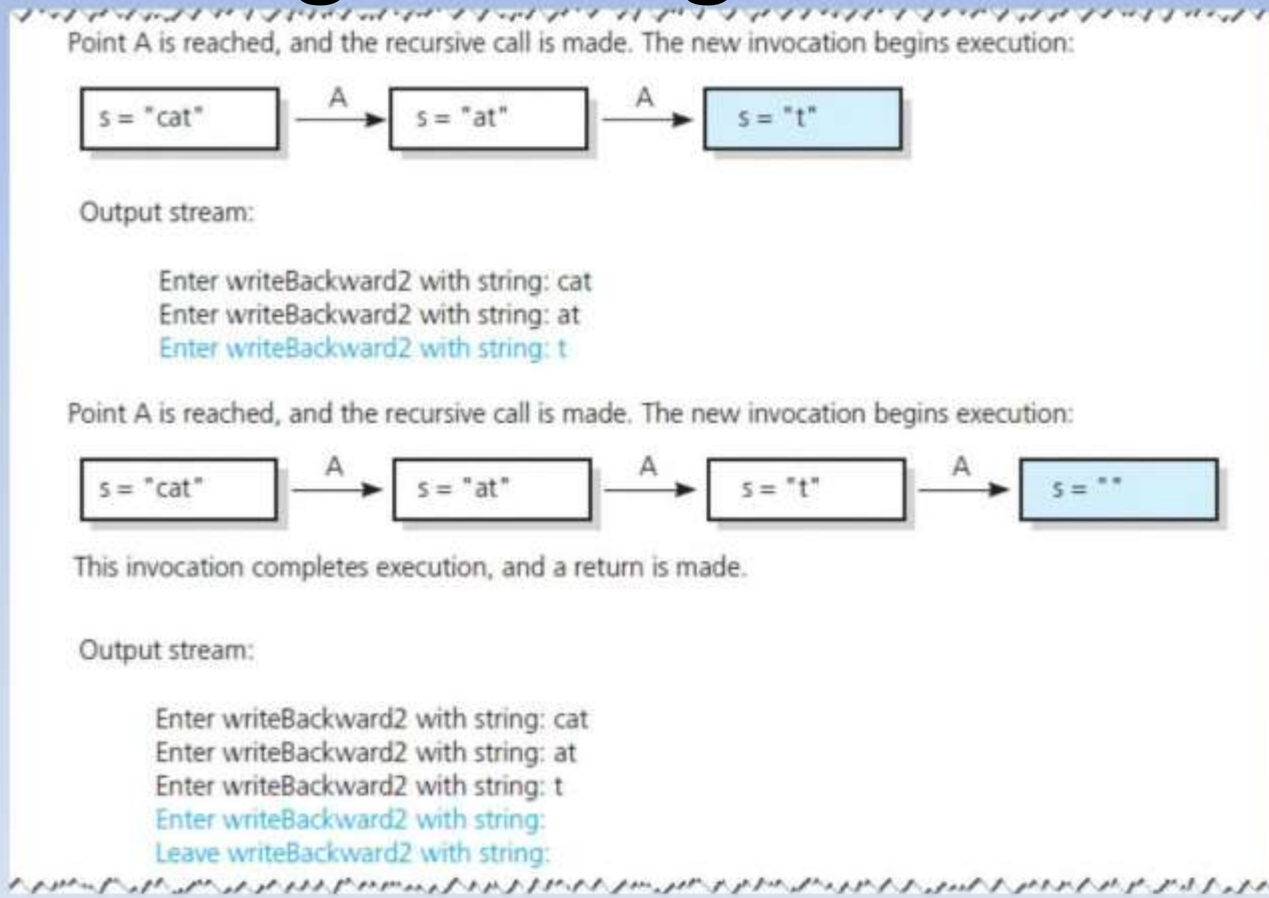


FIGURE 2-8 Box trace of writeBackward2("cat") in pseudocode

A Recursive Void Function: Writing a String Backward

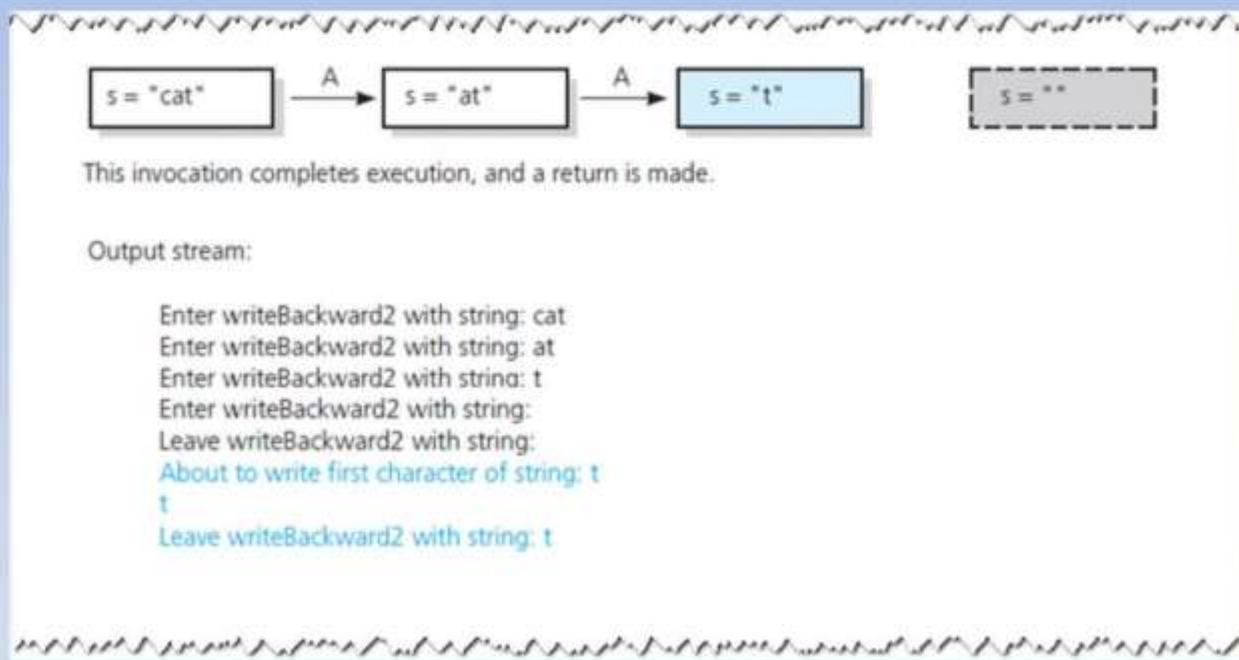


FIGURE 2-8 Box trace of writeBackward2("cat") in pseudocode

A Recursive Void Function: Writing a String Backward

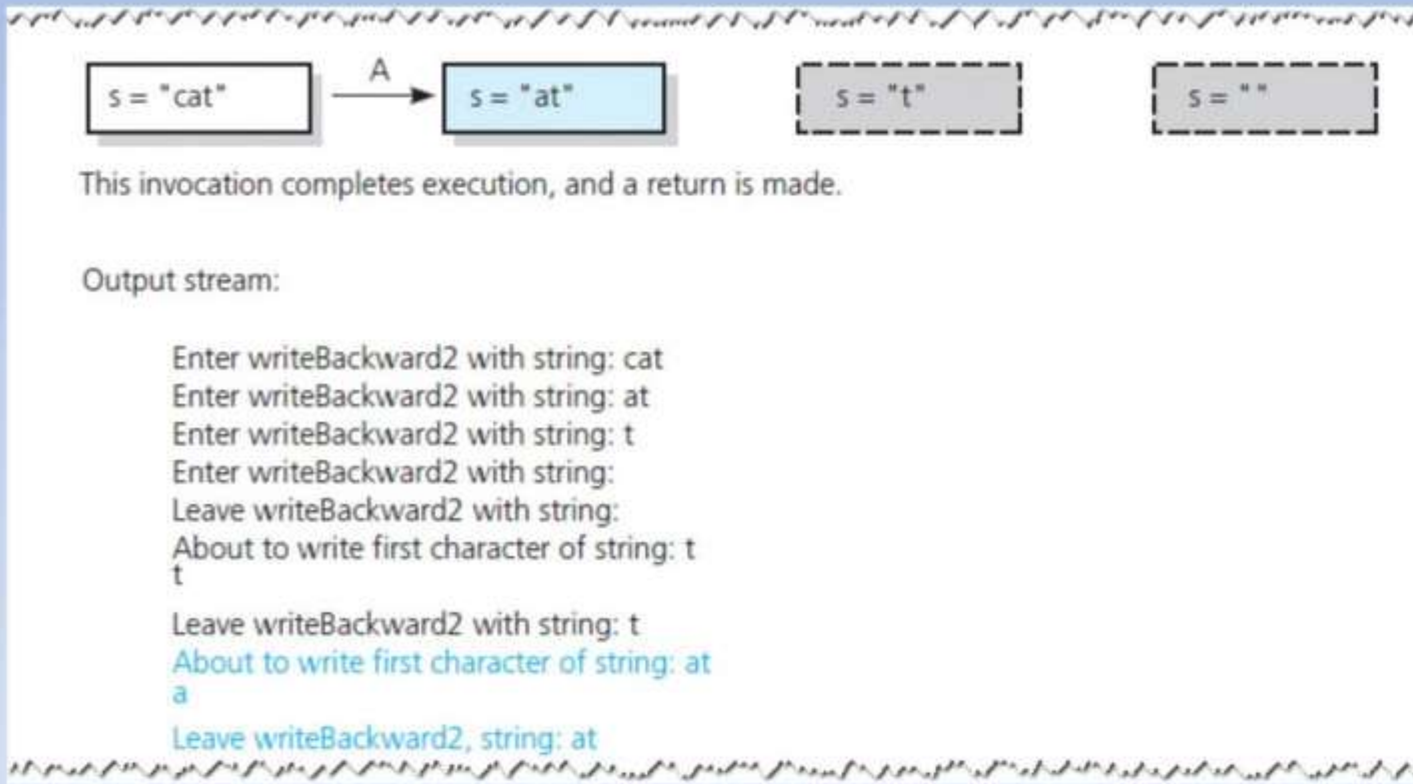


FIGURE 2-8 Box trace of writeBackward2("cat") in pseudocode

A Recursive Void Function: Writing a String Backward

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:
About to write first character of string: t
t
Leave writeBackward2 with string: t
About to write first character of string: at
a
Leave writeBackward2 with string: at
About to write first character of string: cat
c
Leave writeBackward2 with string: cat

FIGURE 2-8 Box trace of writeBackward2("cat") in 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
```

The function writeArrayBackward

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?

The Binary Search

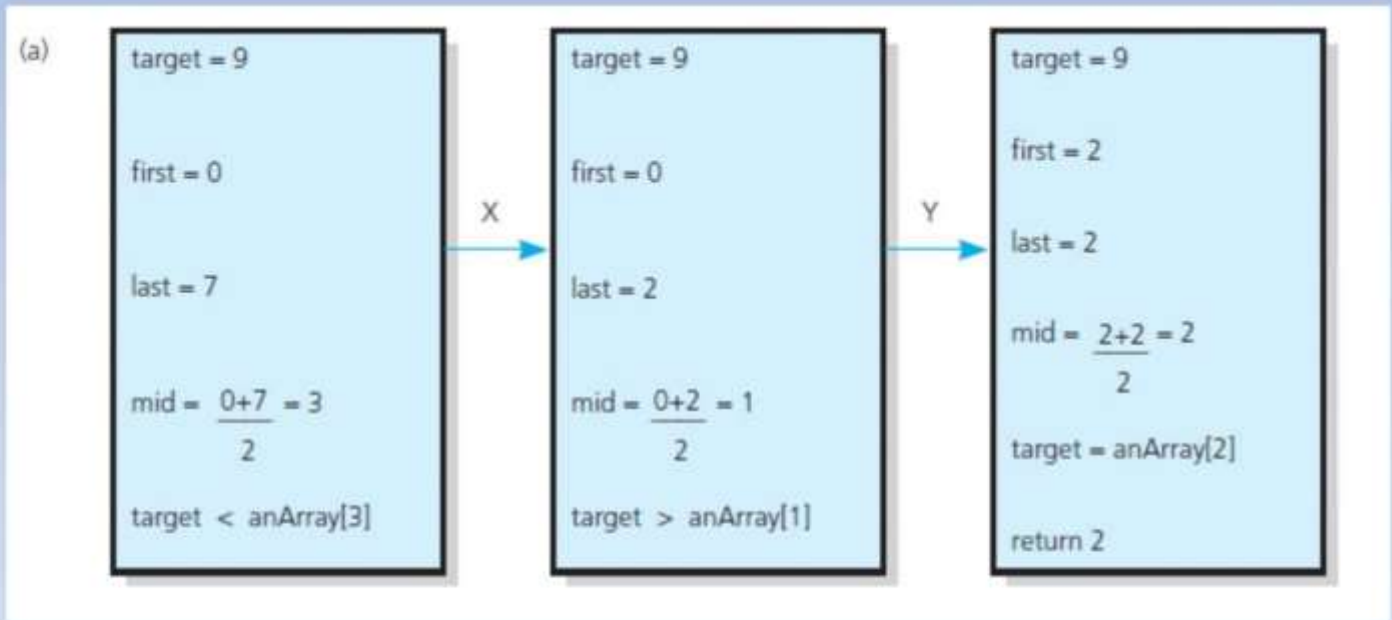


FIGURE 2-10 Box traces of binarySearch with anArray = <1, 5, 9, 12, 15, 21, 29, 31>: (a) a successful search for 9;

The Binary Search

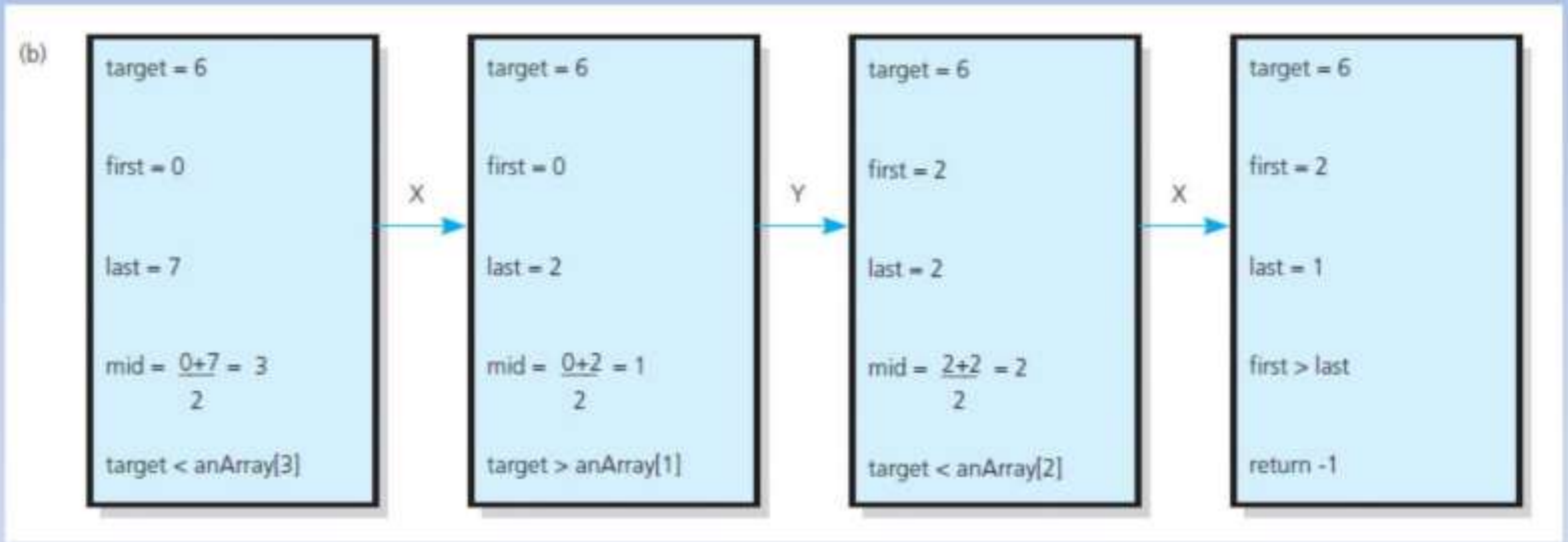


FIGURE 2-10 Box traces of binarySearch with anArray = <1, 5, 9, 12, 15, 21, 29, 31>: (b) an unsuccessful search for 6

The Binary Search

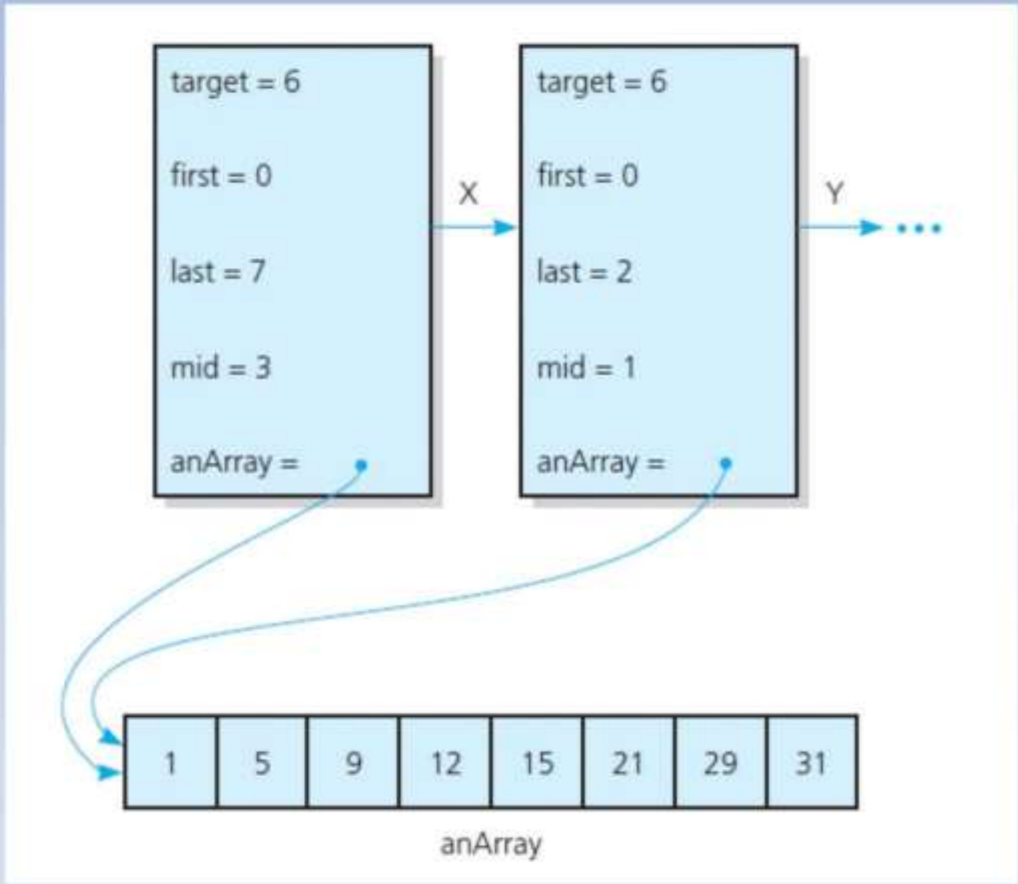


FIGURE 2-11 Box trace with a reference argument

Finding the Largest Value in an Array

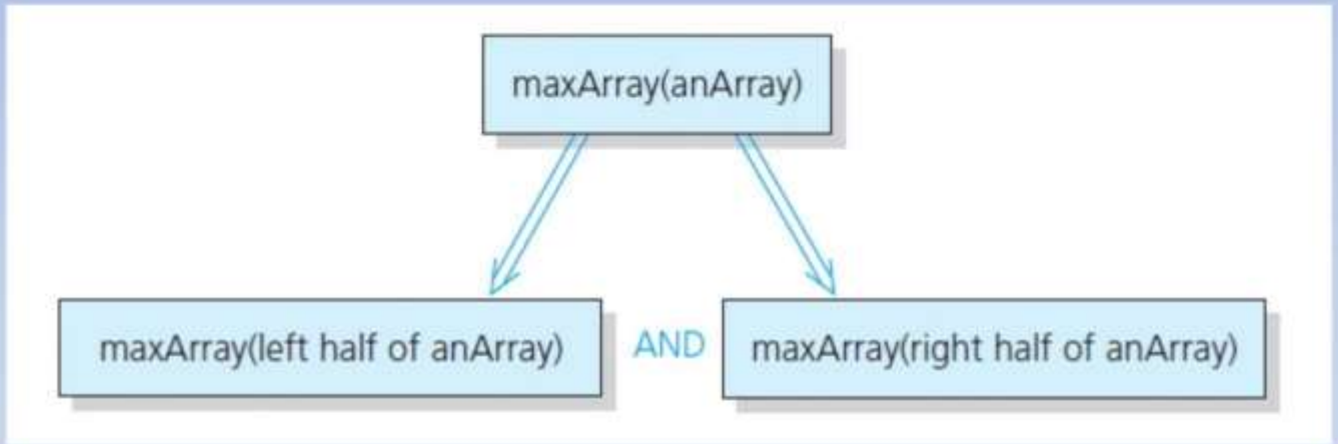


FIGURE 2-12 Recursive solution to the largest-value problem

Finding the Largest Value in an Array

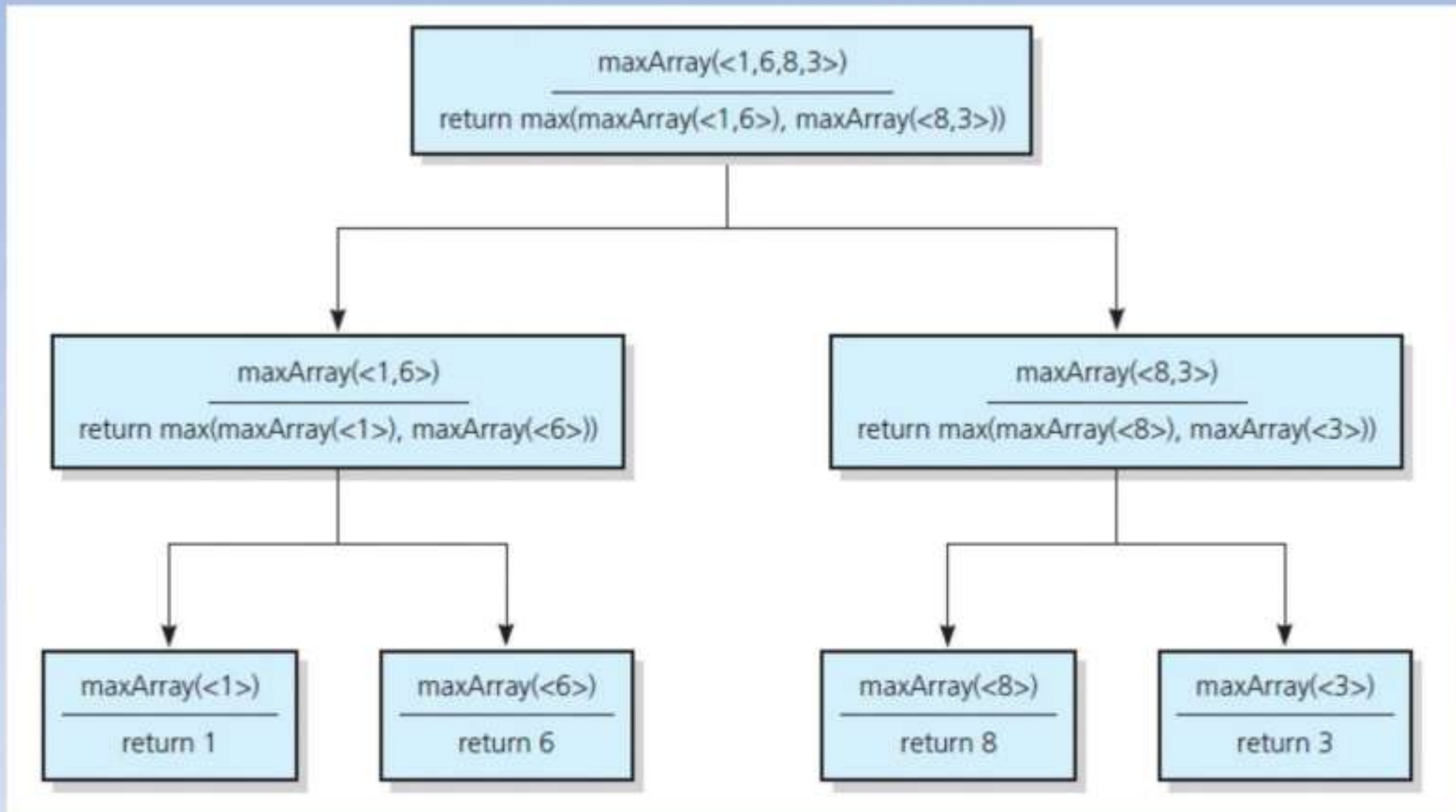


FIGURE 2-13 The recursive calls that maxArray(<1,6,8,3>) generates

Finding kth Smallest Value of Array

Recursive solution proceeds by:

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

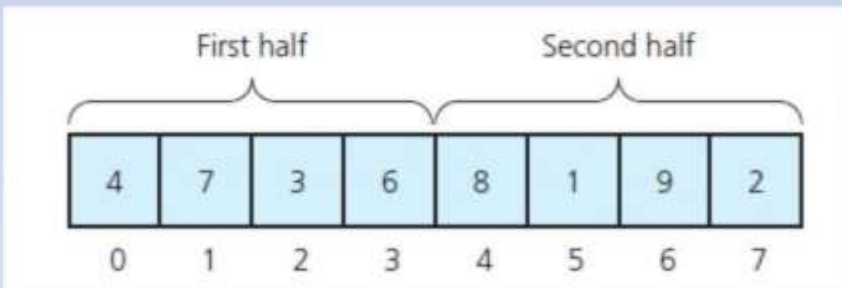


FIGURE 2-14 A sample array

Finding kth Smallest Value of Array

FIGURE 2-15 A partition about a pivot

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)

The Towers of Hanoi

FIGURE 2-16 (

The Towers of Hanoi

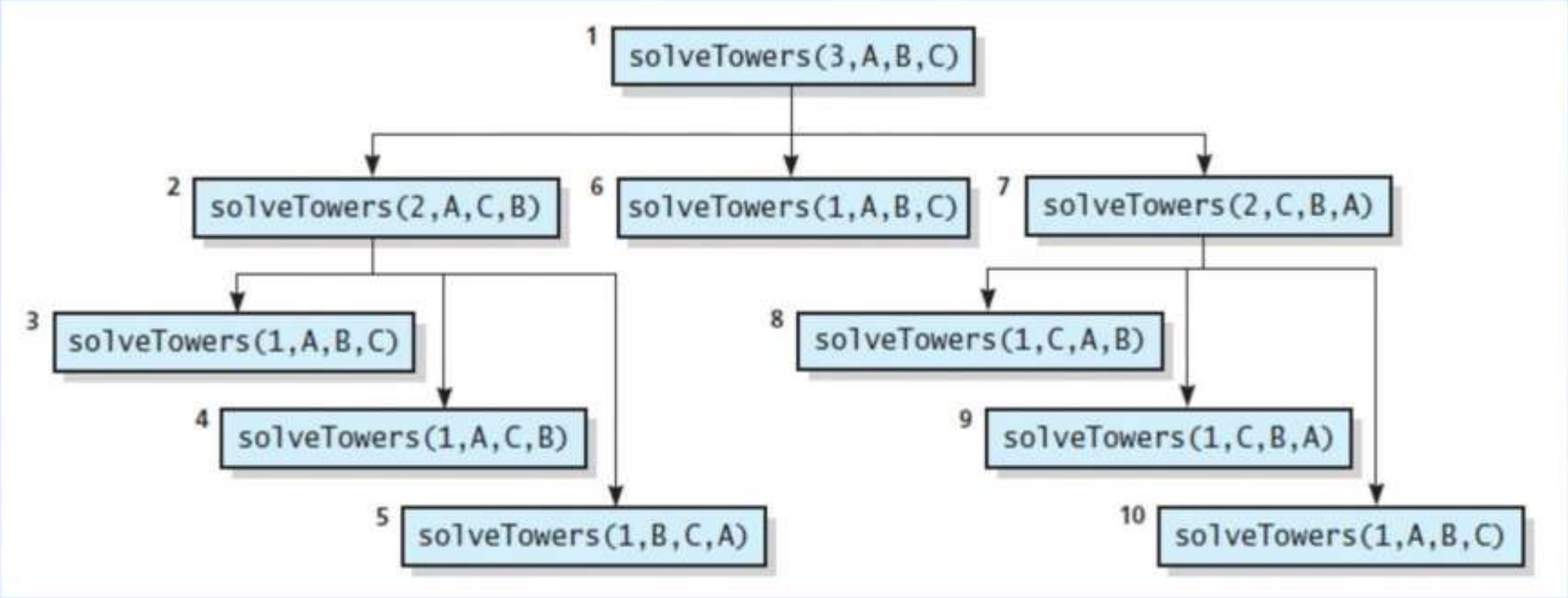


FIGURE 2-17 The order of recursive calls that results from solveTowers(3, A, B, C)

The Fibonacci Sequence (Multiplying Rabbits)

Assume the following “facts”

… •Rabbits never die.

•Rabbit reaches sexual maturity at beginning of third month of life.

•Rabbits always born in male-female pairs. At beginning of every month, each sexually mature male-female pair gives birth to exactly one male-female pair.

The Fibonacci Sequence (Multiplying Rabbits)

Monthly sequence

1. One pair, original two rabbits
2. One pair still
3. Two pairs (original pair, two newborns)
4. Three pairs (original pair, 1 month old, newborns)
5. Five pairs …
6. Eight pairs …

The Fibonacci Sequence (Multiplying Rabbits)

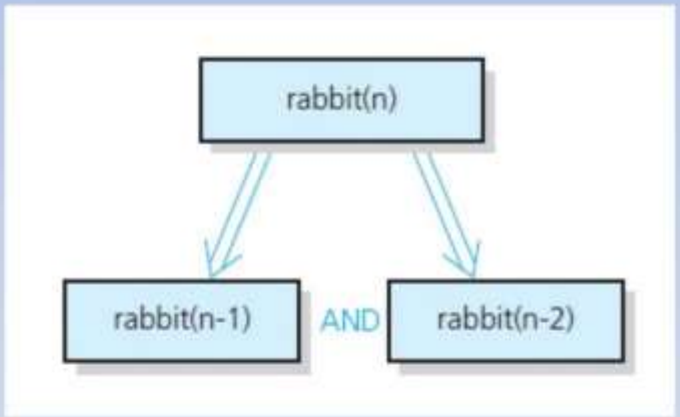
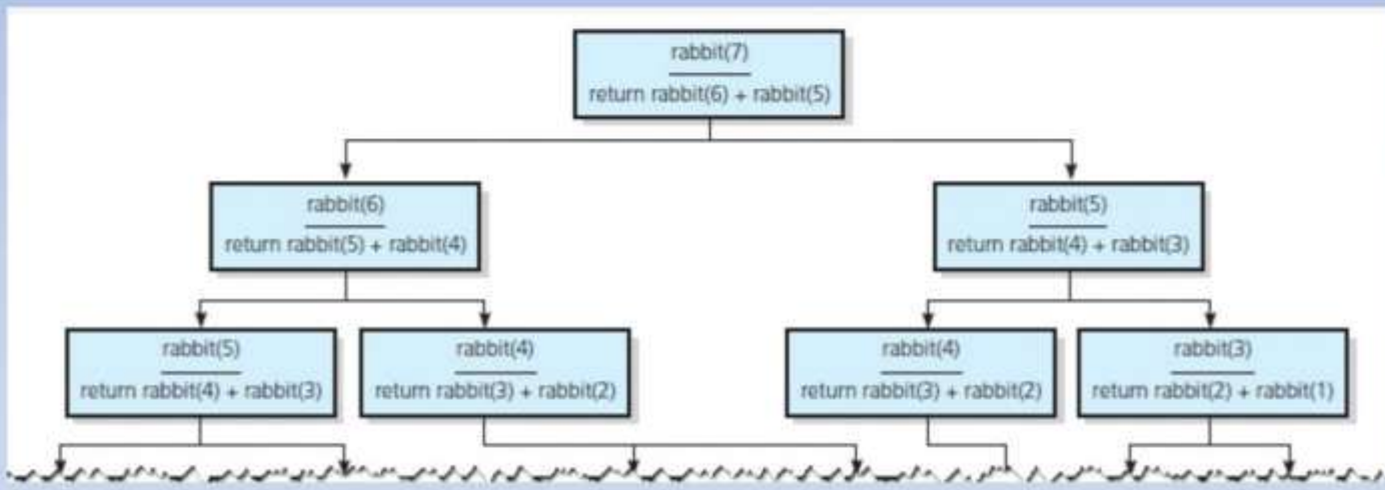


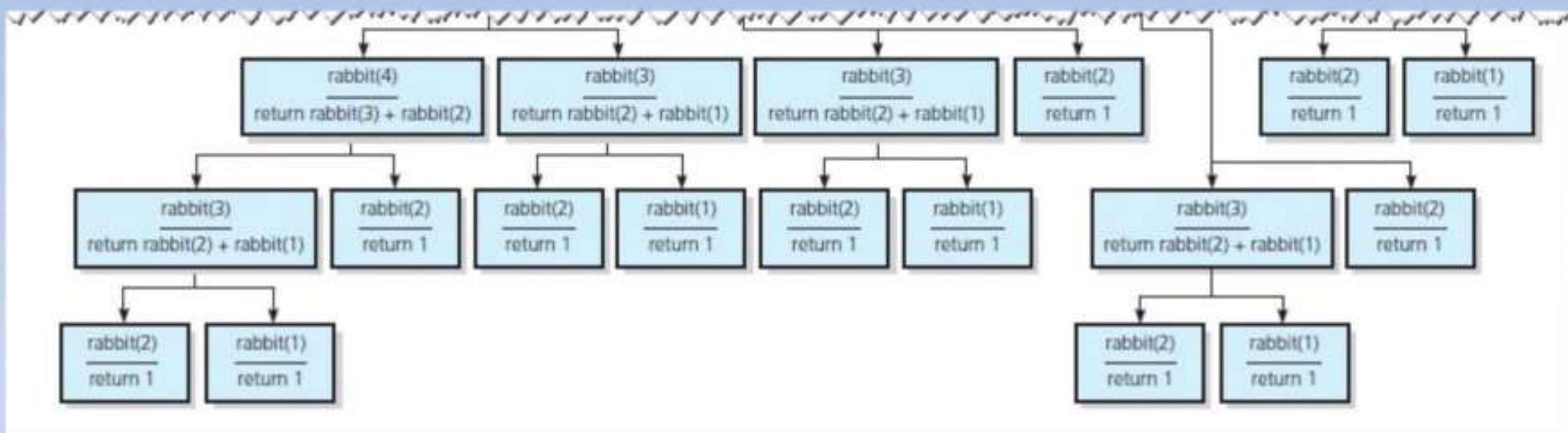
FIGURE 2-18 Recursive solution to the rabbit problem (number of pairs at month *n*)

The Fibonacci Sequence (Multiplying Rabbits)



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

The Fibonacci Sequence (Multiplying Rabbits)



F IGURE 2-19 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

- Possible to see
  - $P(1) = 2$
  - $P(2)=3$
  - $P(n) = P(n-1) + P(n-2)$ for $n > 2$
- 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*

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

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 getNumberOfGroups(n - 1, k - 1) + getNumberOfGroups(n - 1, k);
}  // end getNumberOfGroups
```

Function for recursive solution.

Choosing *k* Out of *n* Things

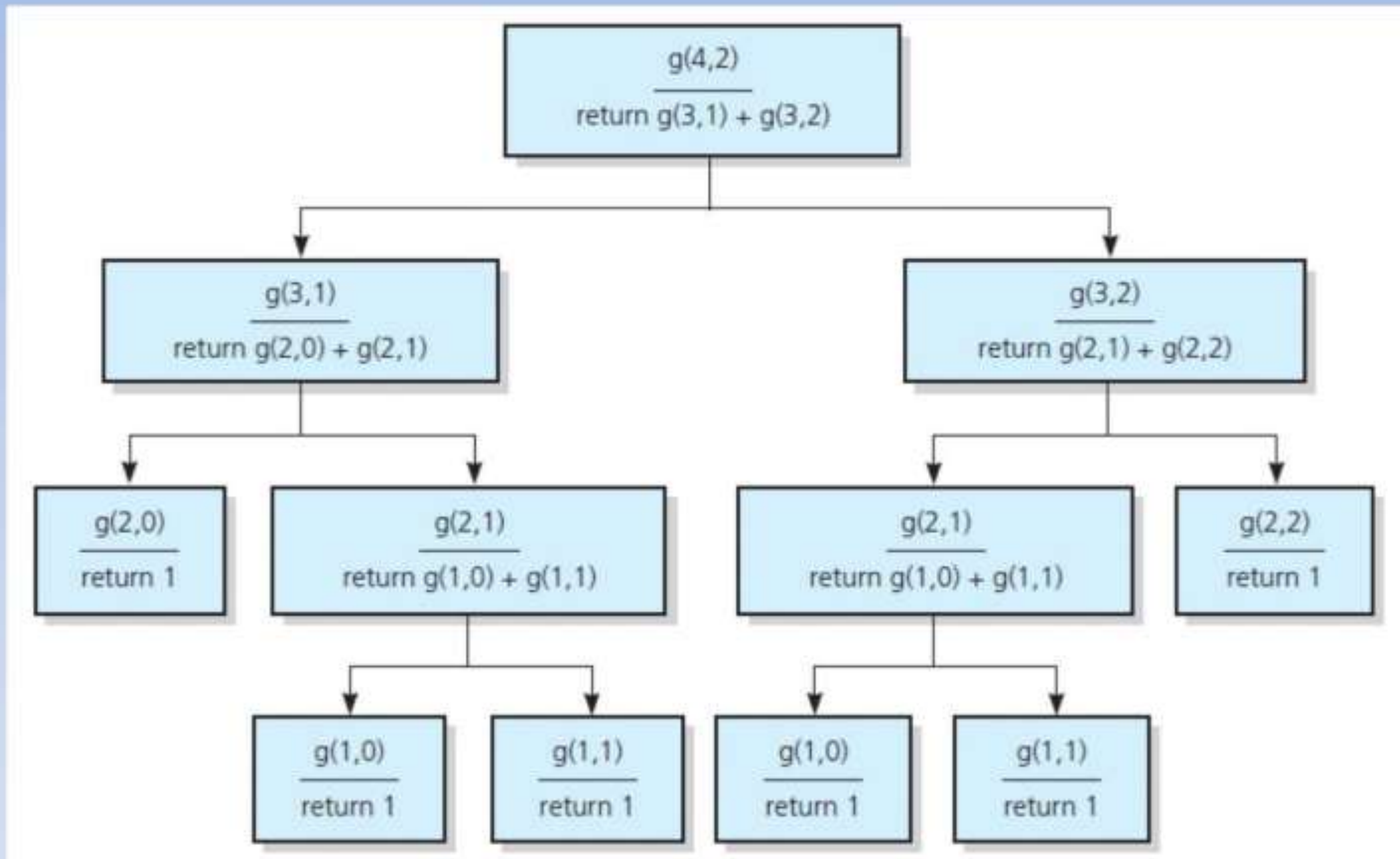


FIGURE 2-20 The recursive calls that g (4, 2) generates

Recursion and Efficiency

- Factors that contribute to inefficiency
  - Overhead associated with function calls
  - Some recursive algorithms inherently inefficient
- Keep in mind
  - Recursion can clarify complex solutions … but …
  - Clear, efficient iterative solution may be better

End

Chapter 2