Recursion

Recursion

- ***Recursion*** is a technique that solves a problem by solving a smaller problem of the same type.
- A ***recursive function*** is a function invoking itself, either directly or indirectly.
- Recursion can be used as an alternative to iteration.
- Recursion is an important and powerful tool in problem solving And programming.
- Recursion is a programming technique that naturally implements the divide-and-conquer problem solving methodology.

The Nature of Recursion

1. One or more simple cases of the problem (called the *stopping cases or base case*) have a simple non-recursive solution.
2. The other cases of the problem can be reduced (*using recursion*) to problems that are closer to stopping cases.
3. Eventually the problem can be reduced to stopping cases only, which are relatively easy to solve.

In general:

if *(stopping case)*

    solve it

else

    reduce the problem using recursion

Four Criteria of A Recursive Solution

1. A recursive function calls itself.
   - This action is what makes the solution recursive.

2. Each recursive call solves an identical, but smaller, problem.
   - A recursive function solves a problem by solving another problem that is identical in nature but smaller in size.

3. A test for the base case enables the recursive calls to stop.
   - There must be a case of the problem (known as *base case* or *stopping case*) that is handled differently from the other cases (without recursively calling itself.)
   - In the base case, the recursive calls stop and the problem is solved directly.

4. Eventually, one of the smaller problems must be the base case.
   - The manner in which the size of the problem diminishes ensures that the base case is eventually is reached.

Four Questions for Constructing Recursive Solutions

1. How can you define the problem in terms of a smaller problem of the same type?

2. How does each recursive call diminish the size of the problem?

3. What instance of the problem can serve as the base case?

4. As the problem size diminishes, will you reach this base case?

Factorial Function – Iterative Definition

n! = n * (n-1) * (n-2) * … * 2 * 1 for any integer n>0

0! = 1

Iterative Definition in C:

```
fval = 1;
for (i = n; i >= 1; i--)
    fval = fval * i;
```

Factorial Function - Recursive Definition

- To define ***n!*** recursively, ***n!*** must be defined in terms of the factorial of a smaller number.
- Observation (problem size is reduced):

  n! = n * (n-1)!
- Base case: 0! = 1
- We can reach the base case, by subtracting 1 from n if n is a positive integer.

Recursive Definition:

$n! = 1$ if $n = 0$

$n! = n*(n-1)!$ if $n > 0$

Factorial Function - Recursive Definition in C

```
// Computes the factorial of a nonnegative integer.
// Precondition: n must be greater than or equal to 0.
// Postcondition: Returns the factorial of n; n is unchanged.
int fact(int n)
{
  if (n ==0)
     return (1);
  else
     return (n * fact(n-1));
}
```

- This ***fact*** function satisfies the four criteria of a recursive solution.

Tracing a Recursive Function

- A **stack** is used to keep track of function calls.
- Whenever a new function is called, all its parameters and local variables are pushed onto the stack along with the memory address of the calling statement (this gives the computer the return point after execution of the function)
  - For each function call, an ***activation record*** is created on the stack.
- To trace a recursive function, the ***box method*** can be used.
  - The box method is a systematic way to trace the actions of a recursive function.
  - The box method illustrates how compilers implement recursion.
  - Each box in the box method roughly corresponds to an activation record.

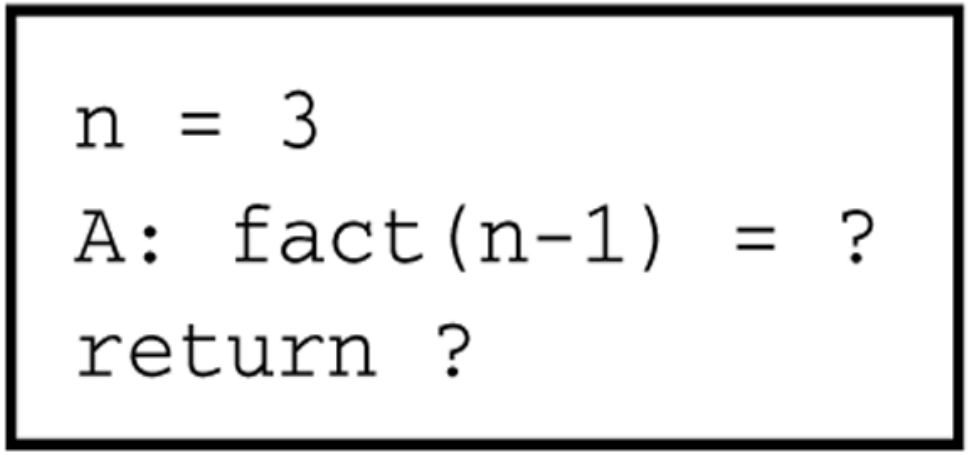
The Box Method (for a valued function)

1. Label each recursive call in the body of the recursive function.
   - For each recursive call, we use a different label to distinguish different recursive calls in the body.
   - These labels help us to keep track of the correct place to which we must return after a function call completes.
   - After each recursive call, we return to the labeled location, and substitute that recursive call with returned valued.

```
if (n ==0)
   return (1);
else
  return (n * fact(n-1) )
                    A
```

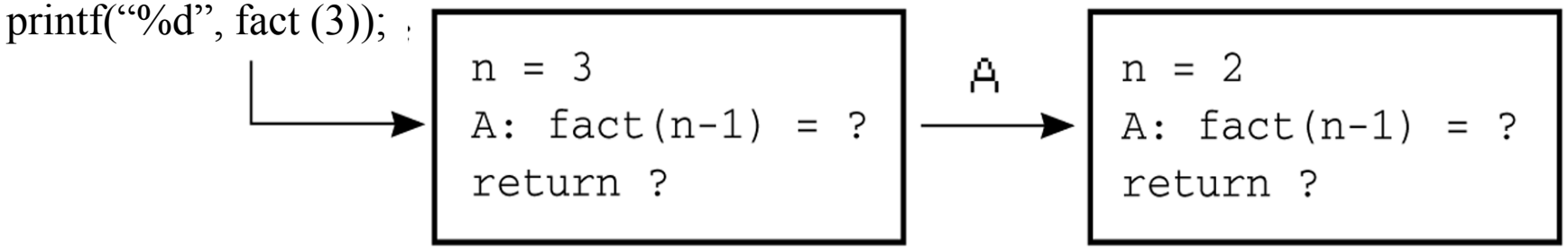
The Box Method (continued)

2. Each time a function is called, a new box represents its local environment. Each box contains:
   - the values of the arguments,
   - the function local variables
   - A placeholder for the value returned from each recursive call from the current box. (label in step 1)
   - The value of the function itself.



The Box Method (continued)

3. Draw an arrow from the statement that initiates the recursive process to the first box.
   - Then draw an arrow to a new box created after a recursive call, put a label on that arrow.



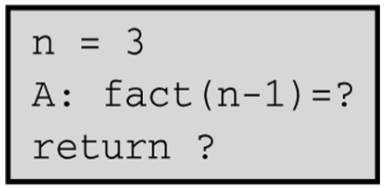
The Box Method (continued)

4. After a new box is created, we start to execute the body of the function.

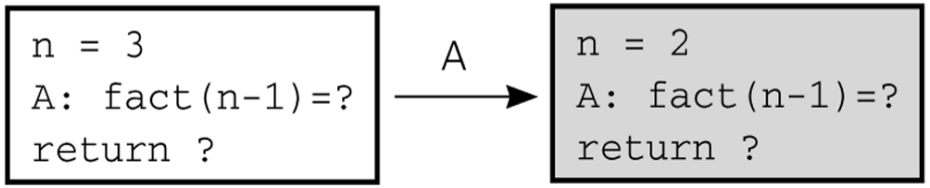
5. On exiting a function, cross off the current box and follow its arrow back to the box that called the function.
   - This box becomes the current box.
   - Substitute the value returned by the just-terminated function call into the appropriate item in the current box.
   - Continue the execution from the returned point.

Box Trace of fact(3)

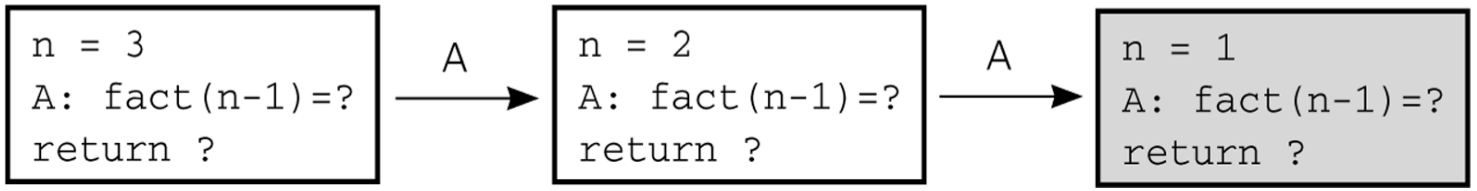
The initial call is made, and method `fact` begins execution:



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

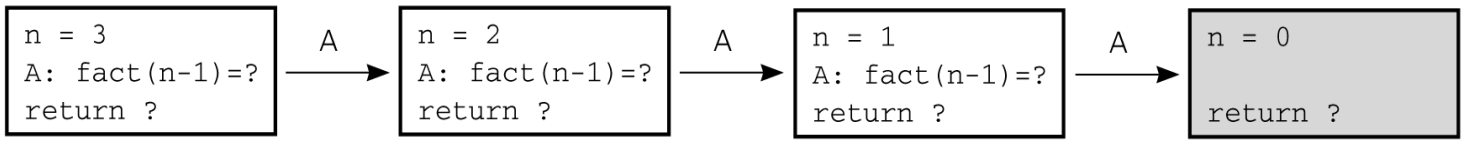


At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

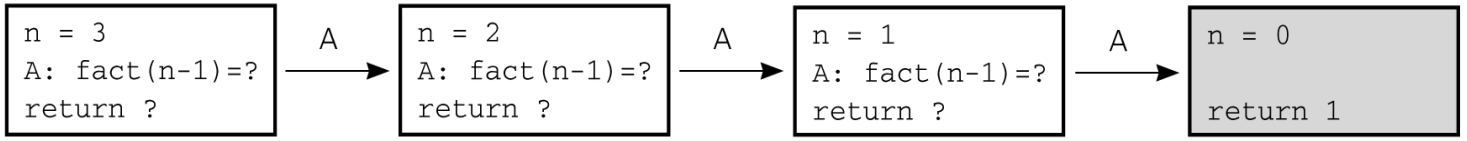


Box Trace of fact(3) (continued)

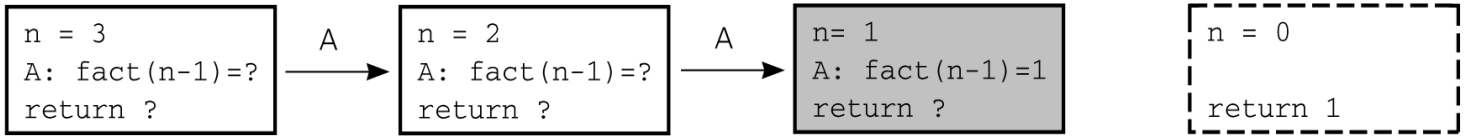
At point A a recursive call is made, and the new invocation of the method `fact` begins execution:



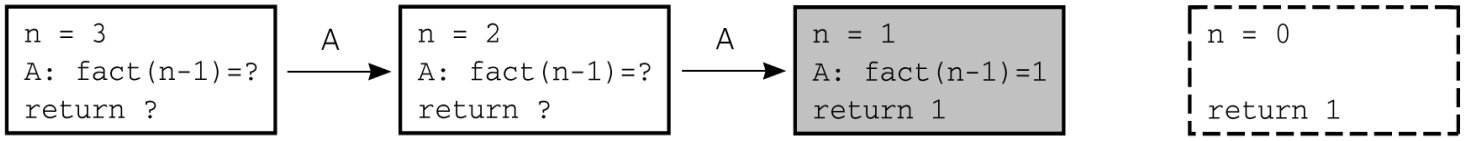
This is the base case, so this invocation of `fact` completes:



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

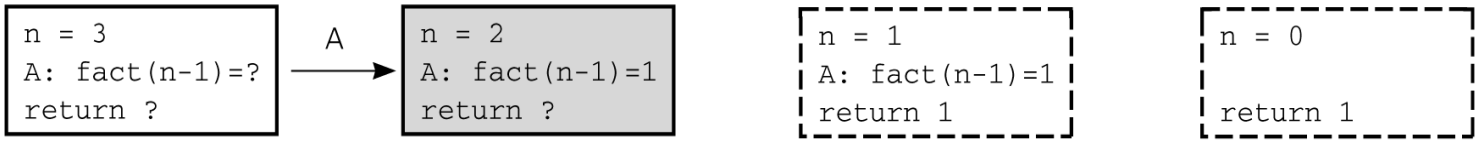


The current invocation of `fact` completes:

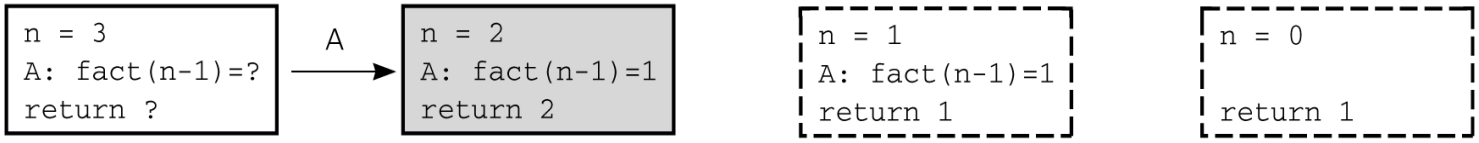


Box Trace of fact(3) (continued)

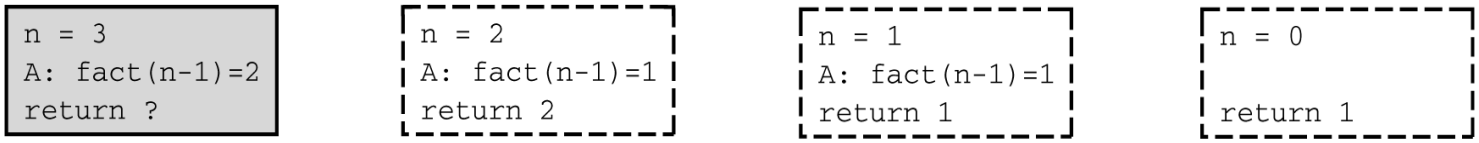
The method value is returned to the calling box, which continues execution:



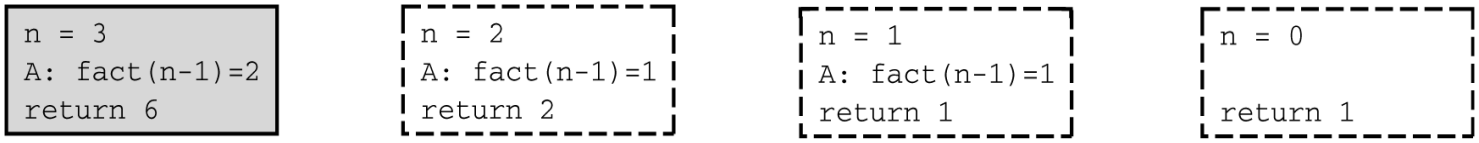
The current invocation of `fact` completes:



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



The current invocation `fact` completes:



The value 6 is returned to the initial call.

A Recursive void Function – Writing a String Backward

Problem: Write the given string of characters in reverse order.

Recursive Solution:

Base Case: Write the empty string backward.

Recursive Case: How can we write an n-character string backward, if we can write an (n-1)-character string backward?

- Strip away the last character, or strip away the first character

writeBackward(in s:string)

if *(the string is empty)*

Do nothing // base case

else *{*

Write the last character of s

writeBackward(s minus its last character)

}

Writing a String Backward in C

```
// Writes a character string backward.
// Precondition: The string s contains size characters, where size >= 0.
// Postcondition: s is written backward, but remains unchanged.
void writeBackward(char s[], int size)
{
   if (size > 0)
   { // write the last character
      printf("%c", s[size-1]);
      // write the rest of the string backward
      writeBackward(s, size-1); // Point A
   } // end if
   // size == 0 is the base case - do nothing
} // end writeBackward
```

Box trace of writeBackward("cat",3)

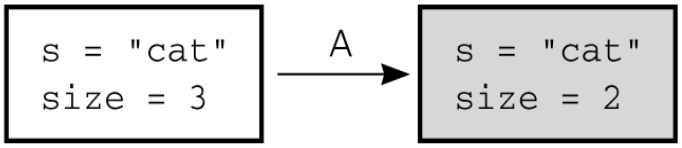
The initial call is made, and the function begins execution:

```
s = "cat"
size = 3
```

Output line: **t**

Point A `(writeBackward(s, size-1))` is reached, and the recursive call is made.

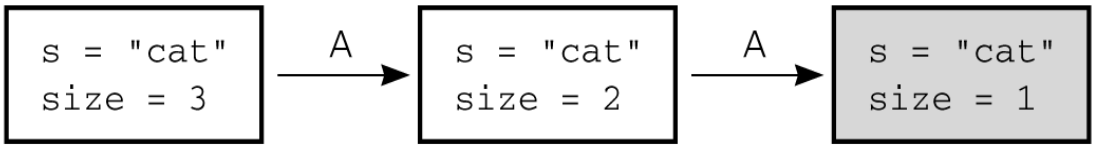
The new invocation begins execution:



Output line: **ta**

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

The new invocation begins execution:

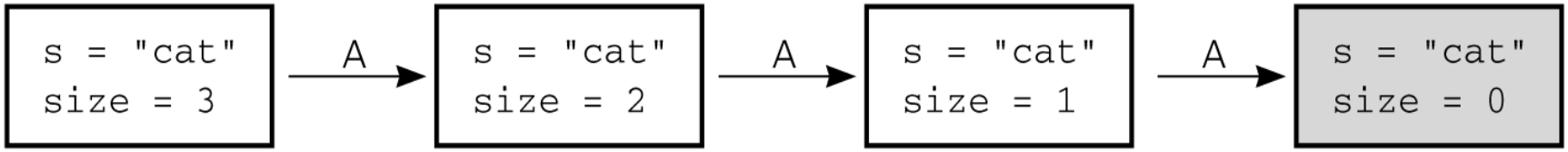


Box trace of writeBackward(“cat”,3) (continued)

Output line: **tac**

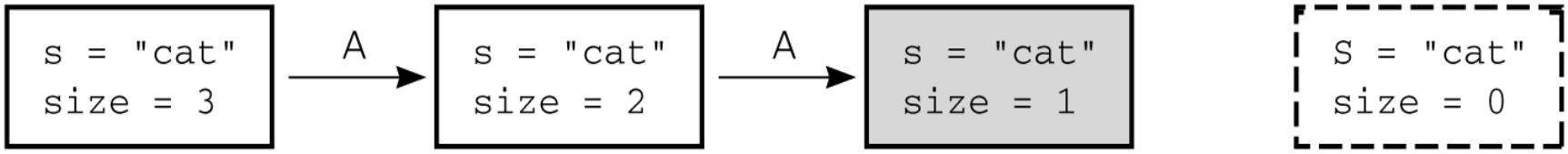
Point A is reached, and the recursive call is made.

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”,3) (continued)

This invocation completes. Control returns to the calling box, which continues execution:

This invocation completes. Control returns to the calling box, which continues execution:

This invocation completes. Control returns to the statement following the initial call.

writeBackward2 in pseudocode

```
writeBackward2 (in s:string)
    if  (the string is empty)
        Do nothing          // base case
    else {
        writeBackward2(s minus its first character)
        Write the first character of s
    }
```

Multiplying Rabbits – The Fibonacci Sequence

- Rabbits give birth so often. If rabbits did not die, their population would be quickly get out of hand.
- Let us assume that:
  - Rabbits never die.
  - A rabbit reaches sexual maturity exactly two months after birth (at the beginning of its third month of life).
  - Rabbits are always born male-female pairs.
  - At the beginning of every month, each sexually mature male-female pair gives birth to exactly one male-female pair.

- ***Question***:
  - Suppose we start with a single newborn male-female pair in the first month.
  - What will be the number rabbit pairs in month **n**?

Multiplying Rabbits – First Seven Months

Month 1: 1 pair

Month 2: 1 pair

- since it is not yet sexually mature

Month 3: 2 pairs

- 1 original pair + a newborn pair from the original pair because it is now sexually mature.

Month 4: 3 pairs

- 2 pairs alive in month 3 + a newborn pair from original pair.

Month 5: 5 pairs

- 3 pairs alive in month 4 + 2 new newborn pairs from 2 pairs alive in month 3.

Month 6: 8 pairs

- 5 pairs alive in month 5 + 3 new newborn pairs from 3 pairs alive in month 4.

Month 7: 13 pairs

- 8 pairs alive in month 6 + 5 new newborn pairs from 5 pairs alive in month 5.

Recursive Solution to Rabbit Problem

Observation:

- All of the pairs alive in month n-1 cannot give birth at the beginning of month n.
- Only, all of the pairs alive in month n-2 can give birth.
- The number pairs in month n is the sum of the number of pairs alive in month n-1 plus the number rabbits alive in month n-2.

Recurrence relation for the number of pairs in month n:

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

Recursive Solution to Rabbit Problem

- Two base cases are necessary because there are two smaller problems.
  - rabbit(1) = 1 rabbit(2) = 1

Recursive Solution:

rabbit(n) = 1 *if n is 1 or 2*

rabbit(n) = rabbit(n-1) + rabbit(n-2) *if n > 2*

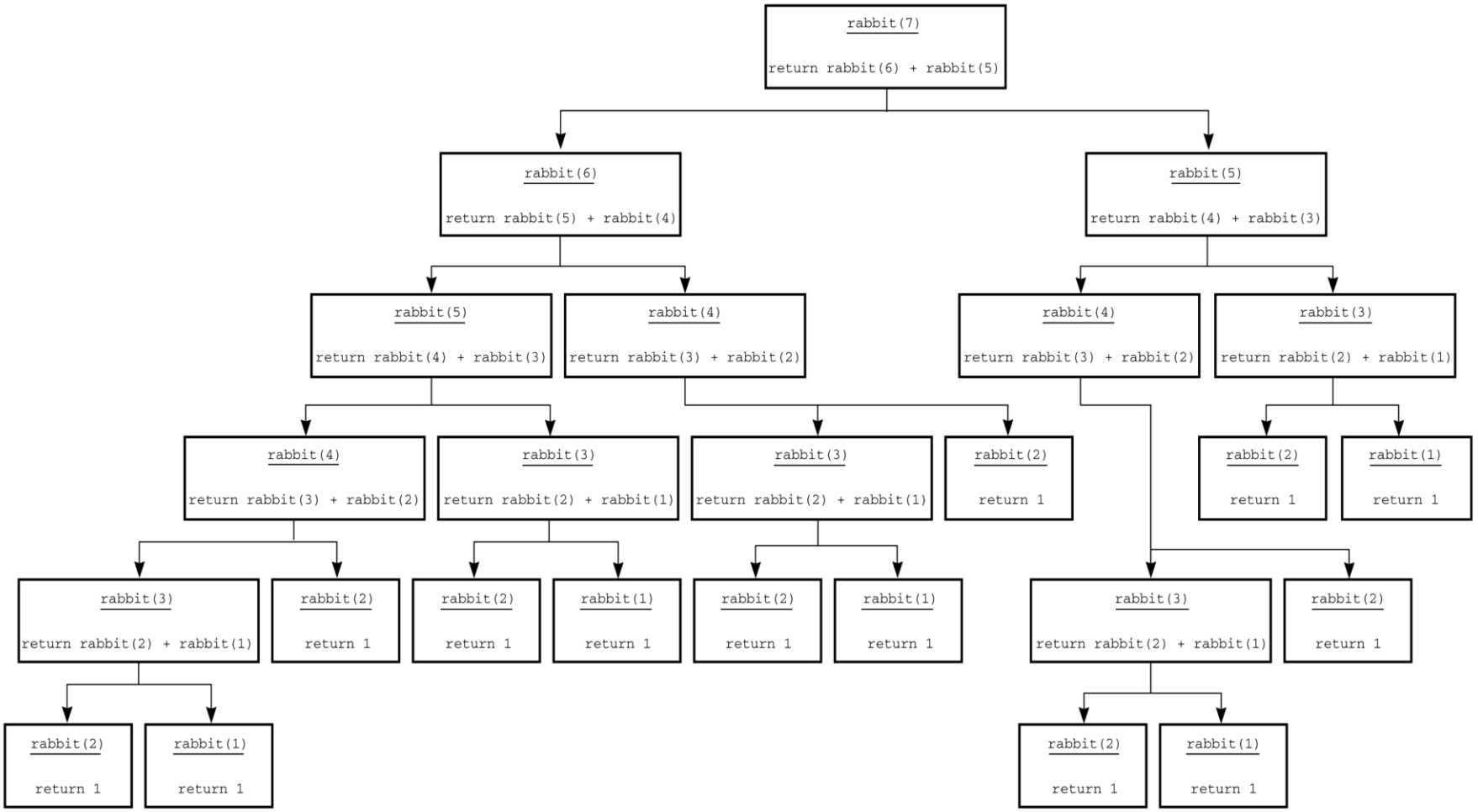
- The series of numbers rabbit(1), rabbit(2), rabbit(3), … is known as **Fibonacci Sequence**.

Recursive Solution to Rabbit Problem in C

```
// Computes a term in the Fibonacci sequence.
// Precondition: n is a positive integer.
// Postcondition: Returns 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
```

- *This rabbit function computes Fibonacci sequence (but inefficiently).*

Recursive Calls for rabbit(7)



Binary Search

Problem: Search a sorted array of integers for a given value.

Recursive Binary Search Algorithm:

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

Binary Search in C

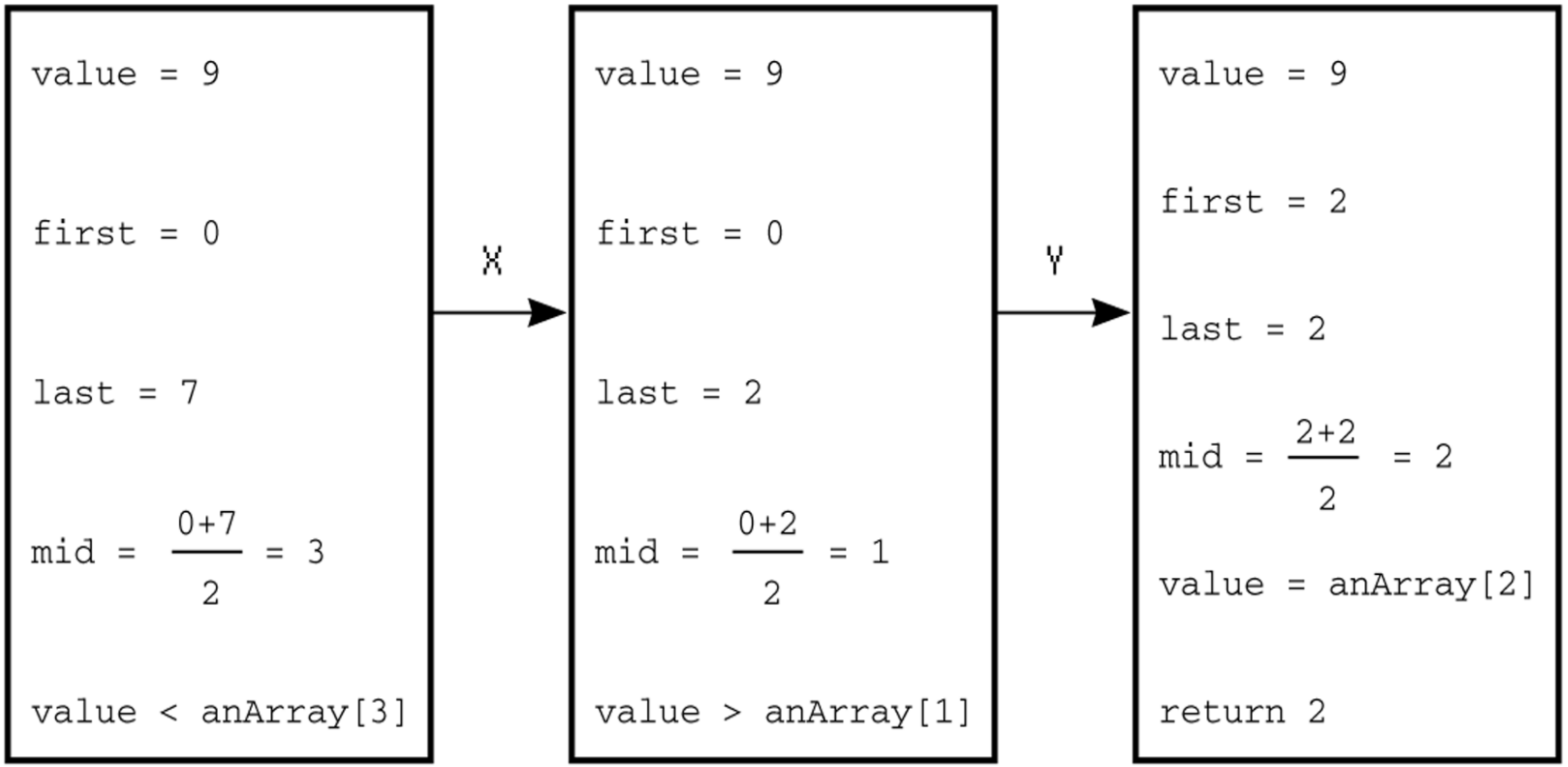
```
int binarySearch(int anArray[], int first, int last, int value)
// Searches the array items anArray[first] through anArray[last] for value by using a binary search.
// Precondition: 0 <= first, last <= SIZE-1, where SIZE is the maximum size of the array, and
// anArray[first] <= anArray[first+1] <= ... <= anArray[last].
// Postcondition: If value is in the array, the function returns the index of the array item that equals value;
// otherwise the function returns -1.
{ int index;
  if (first > last)   index = -1; // value not in original array
  else { // Invariant: If value is in anArray, anArray[first] <= value <= anArray[last]
    int mid = (first + last)/2;
    if (value == anArray[mid])
      index = mid;   // value found at anArray[mid]
    else if (value < anArray[mid])
      index = binarySearch(anArray, first, mid-1, value);  // point X
    else
      index = binarySearch(anArray, mid+1, last, value);  // point Y
  } // end else
  return index;
} // end binarySearch
```

Box Trace of binarySearch (successful)

Box traces of binarySearch with anArray = <1, 5, 9, 12, 15, 21, 29, 31>

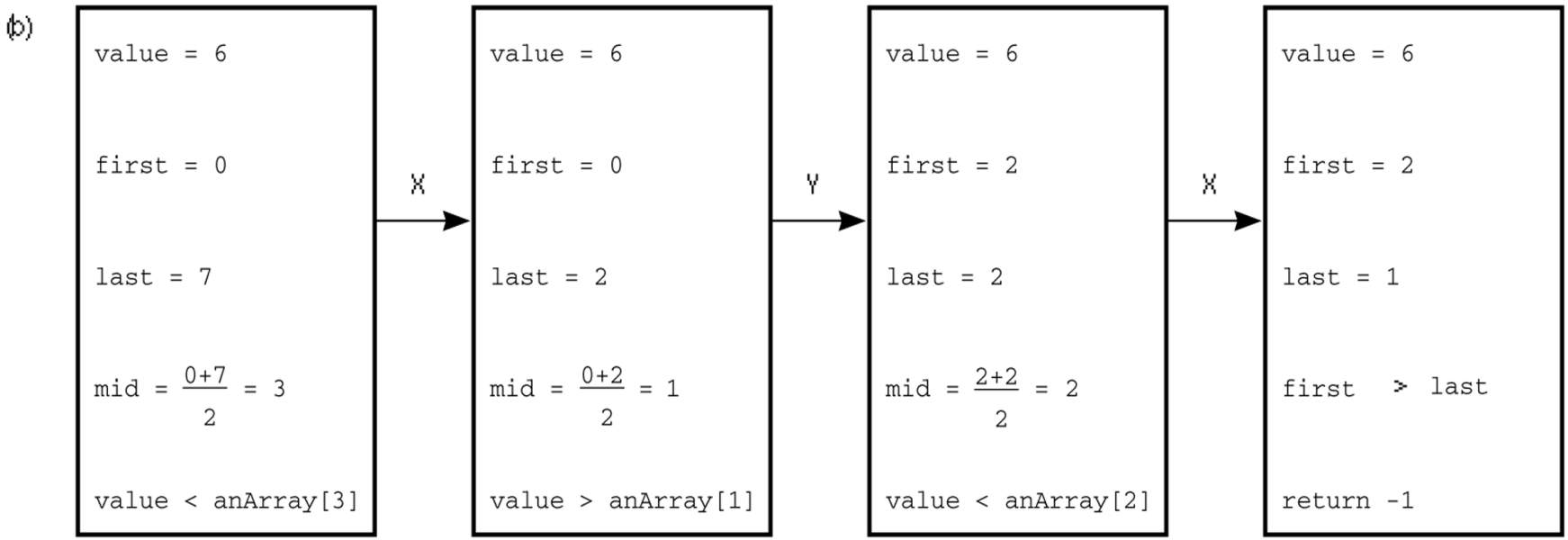
➔ a successful search for 9

(a)

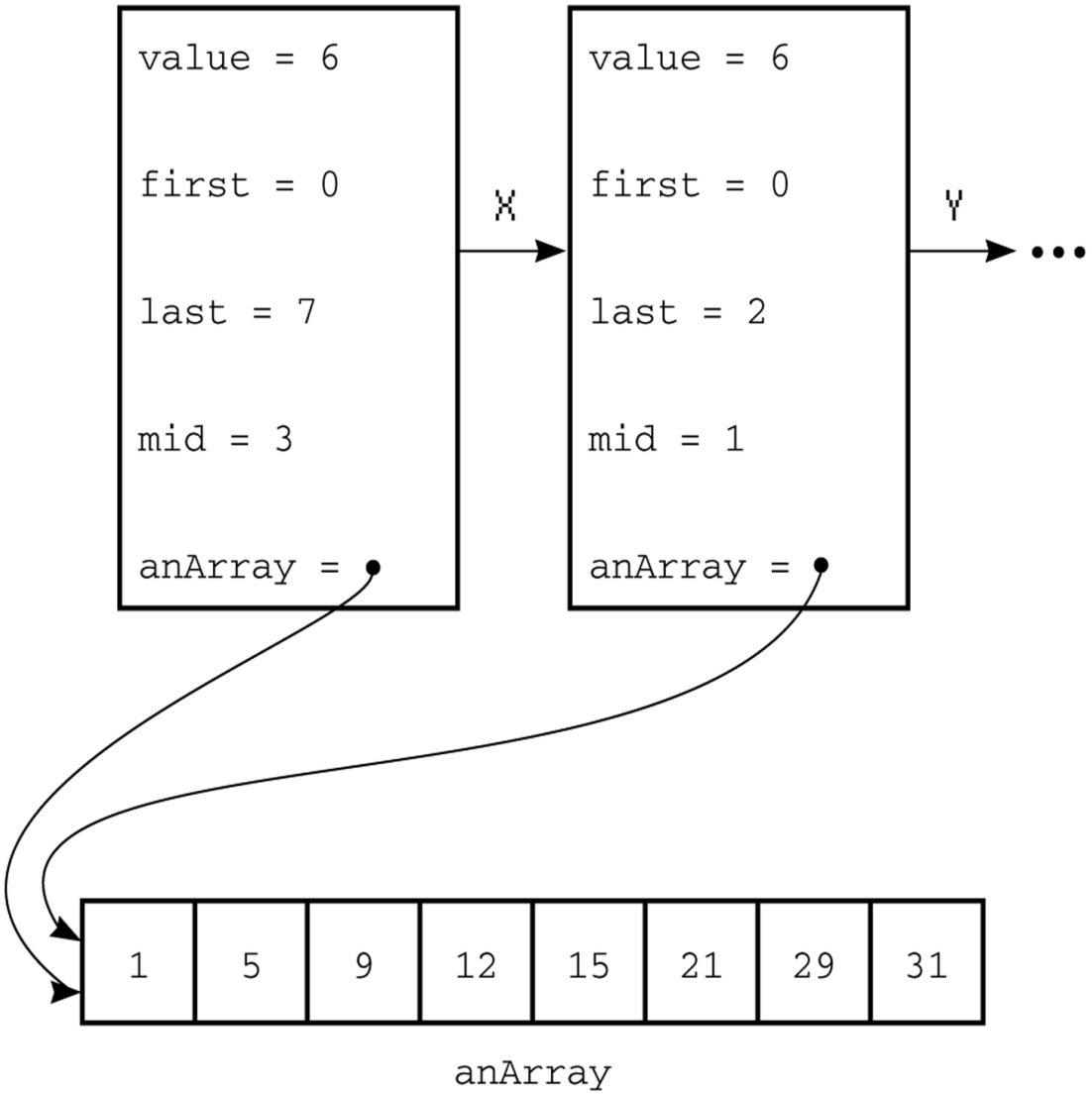


Box Trace of binarySearch (unsuccessful)

Box traces of binarySearch with anArray = <1, 5, 9, 12, 15, 21, 29, 31>
➔ an unsuccessful search for 6



Box Method with A Reference Argument



Growth-Rate Functions – Recursive Algorithms

```
void hanoi(int n, char source, char dest, char spare) {      Cost
  if (n > 0) {                                                c1
    hanoi(n-1, source, spare, dest);                          c2
     cout << "Move top disk from pole " << source             c3
          << " to pole " << dest << endl;
    hanoi(n-1, spare, dest, source);                          c4
} }
```

- The time-complexity function T(n) of a recursive algorithm is defined in terms of itself, and this is known as **recurrence equation** for T(n).
- To find the growth-rate function for a recursive algorithm, we have to solve its recurrence relation.

Growth-Rate Functions – Hanoi Towers

- What is the cost of `hanoi(n,'A','B','C')`?

when n=0

T(0) = c1

when n>0

T(n) = c1 + c2 + T(n-1) + c3 + c4 + T(n-1)

= 2*T(n-1) + (c1+c2+c3+c4)

= **2*T(n-1) + c** ← recurrence equation for the growth-rate function of hanoi-towers algorithm

- Now, we have to solve this recurrence equation to find the growth-rate function of hanoi-towers algorithm

Growth-Rate Functions – Hanoi Towers (cont.)

- We will use a simple method known as *repeated substitutions* to solve recurrence equations.

$T(n) = 2*T(n-1) + c$

$= 2 * (2*T(n-2)+c) + c$

$= 2 * (2* (2*T(n-3)+c) + c) + c$

$= 2^3 * T(n-3) + (2^2+2^1+2^0)*c$ (assuming n>2)

when substitution repeated i-1th times

$= 2^i * T(n-i) + (2^{i-1}+ ... +2^1+2^0)*c$

when i=n

$= 2^n * T(0) + (2^{n-1}+ ... +2^1+2^0)*c$

$= 2^n * c1 + (\sum_{i=0}^{n-1} 2^i)*c$

$= 2^n * c1 + (2^n-1)*c = 2^n*(c1+c) – c$ ➔ So, the growth rate function is **O(2^n)**