Data Structures and Algorithms

Chapter 6
Recursion

Objectives

- Learn about recursive definitions
- Explore the base case and the general case of a recursive definition
- Learn about recursive algorithm
- Learn about recursive functions
- Explore how to use recursive functions to implement recursive

Recursive Definitions

- Recursion
  - Process of solving a problem by reducing it to smaller versions of itself
- Example: factorial problem
  - 5!
    - 5 x 4 x 3 x 2 x 1 =120
  - If *n* is a nonnegative
    - Factorial of *n* (*n*!) defined as follows:

$$0! = 1 \qquad \text{(Equation 6-1)}$$

$$n! = n \times (n-1)! \quad \text{if} \quad n > 0 \qquad \text{(Equation 6-2)}$$

Recursive Definitions (cont'd.)

- Direct solution (Equation 6-1)
  - Right side of the equation contains no factorial notation
- Recursive definition
  - A definition in which something is defined in terms of a smaller version of itself
- Base case (Equation 6-1)
  - Case for which the solution is obtained directly
- General case (Equation 6-2)
  - Case for which the solution is obtained indirectly using recursion

General format for many recursive functions

```
if   (some condition for which answer is known)
                                          // base case
     solution statement

else                                      // general case

     recursive function call
```

SOME EXAMPLES . . .

Recursive Definitions (cont'd.)

- Recursive function implementing the factorial function

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

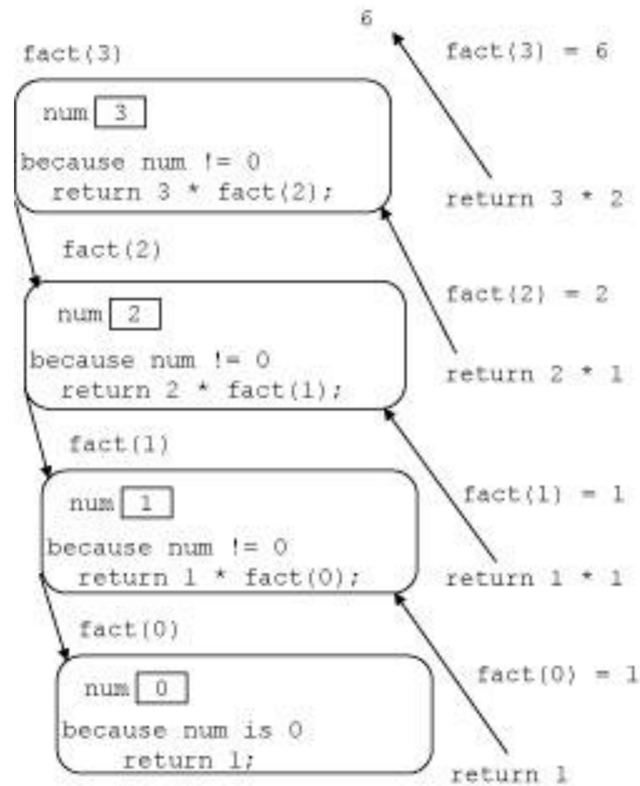

FIGURE 6-1 Execution of fact(4)

Recursive Definitions (cont'd.)

- Recursive function notable comments
  - Recursive function has unlimited number of copies of itself (logically)
  - Every call to a recursive function has its own
    - Code, set of parameters, local variables
  - After completing a particular recursive call
    - Control goes back to calling environment (previous call)
    - Current (recursive) call must execute completely before control goes back to the previous call
    - Execution in previous call begins from point immediately following the recursive call

Recursive Definitions (cont'd.)

- Direct and indirect recursion
  - Directly recursive function
    - Calls itself
  - Indirectly recursive function
    - Calls another function, eventually results in original function call
    - Requires same analysis as direct recursion
    - Base cases must be identified, appropriate solutions to them provided
    - Tracing can be tedious
  - Tail recursive function
    - Last statement executed: the recursive call

Recursive Definitions (cont'd.)

- Infinite recursion
  - Occurs if every recursive call results in another recursive call
  - Executes forever (in theory)
  - Call requirements for recursive functions
    - System memory for local variables and formal parameters
    - Saving information for transfer back to right caller
  - Finite system memory leads to
    - Execution until system runs out of memory
    - Abnormal termination of infinite recursive function

Recursive Definitions (cont'd.)

- Requirements to design a recursive function
  - Understand problem requirements
  - Determine limiting conditions
  - Identify base cases, providing direct solution to each base case
  - Identify general cases, providing solution to each general case in terms of smaller versions of itself

Largest Element in an Array

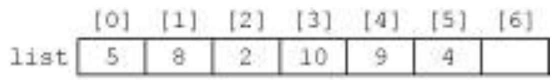

FIGURE 6-2 `list` with six elements

- `list`: array name containing elements
- `list[a]...list[b]` stands for the array elements `list[a], list[a + 1], ..., list[b]`
- `list` length =1
  – One element (largest)
- `list` length >1

```
maximum(list[a],
           largest(list[a + 1]...list[b]))
```

Largest Element in an Array (cont'd.)

- `maximum(list[0], largest(list[1]...list[5]))`
- `maximum(list[1], largest(list[2]...list[5]))`, etc.
- Every time previous formula used to find largest element in a sublist
  - Length of sublist in next call reduced by one

Largest Element in an Array (cont'd.)

- Recursive algorithm in pseudocode

Base Case: The size of the list is 1

The only element in the list is the largest element

General Case: The size of the list is greater than 1

To find the largest element in list[a]...list[b]

```
1. Find the largest element in list[a + 1]...list[b]
   and call it max
2. Compare the elements list[a] and max
   if (list[a] >= max)
      the largest element in list[a]...list[b] is list[a]
   otherwise
      the largest element in list[a]...list[b] is max
```

Largest Element in an Array (cont'd.)

- Recursive algorithm as a C++ function

```
int largest(const int list[], int lowerIndex, int upperIndex)
{
    int max;

    if (lowerIndex == upperIndex) //size of the sublist is one
        return list[lowerIndex];
    else
    {
        max = largest(list, lowerIndex + 1, upperIndex);

        if (list[lowerIndex] >= max)
            return list[lowerIndex];
        else
            return max;
    }
}
```

Largest Element in an Array (cont'd.)

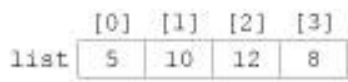

FIGURE 6-3 `list` with four elements

- Trace execution of the following statement

  ```
  cout << largest(list, 0, 3) << endl;
  ```

- Review C++ program on page 362
  - Determines largest element in a list

Largest Element in an Array (cont'd.)

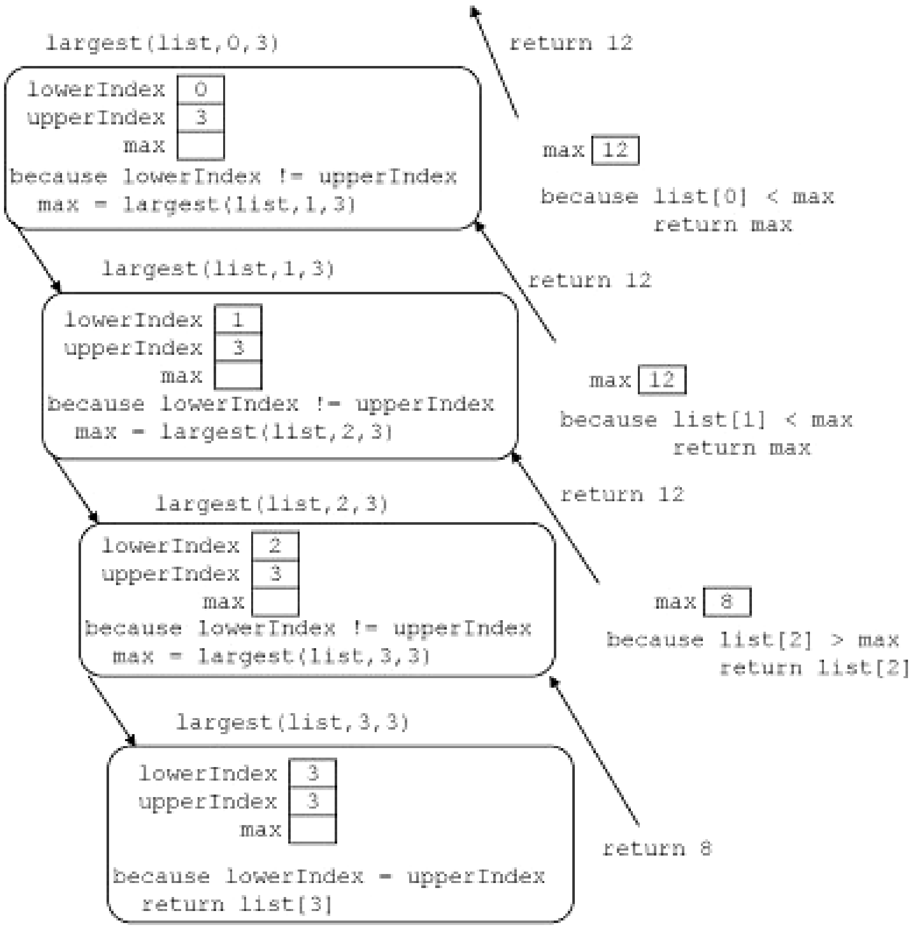

FIGURE 6-4 Execution of `largest(list, 0, 3)`

Print a Linked List in Reverse Order

- Function `reversePrint`
  - Given list pointer, prints list elements in reverse order
- Figure 6-5 example
  - Links in one direction
  - Cannot traverse backward starting from last node

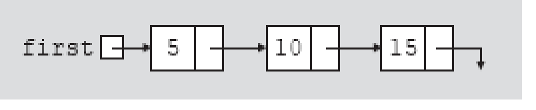

FIGURE 6-5 Linked list

Print a Linked List in Reverse Order (cont'd.)

- Cannot print first node info until remainder of list printed
- Cannot print second node info until tail of second node printed, etc.
- Every time tail of a node considered
  - List size reduced by one
  - Eventually list size reduced to zero
  - Recursion stops

Print a Linked List in Reverse Order (cont'd.)

- Recursive algorithm in pseudocode

Base Case: List is empty: no action

General Case: List is nonempty

1. Print the tail
2. Print the element

- Recursive algorithm in C++

```
if (current != NULL)
{
   reversePrint(current->link);       //print the tail
   cout << current->info << endl;     //print the node
}
```

Print a Linked List in Reverse Order (cont'd.)

- Function template to implement previous algorithm and then apply it to a list

```
template <class Type>
void linkedListType<Type>::reversePrint
                           (nodeType<Type> *current) const
{
    if (current != NULL)
    {
        reversePrint(current->link);    //print the tail
        cout << current->info << " ";   //print the node
    }
}
```

Print a Linked List in Reverse Order (cont'd.)

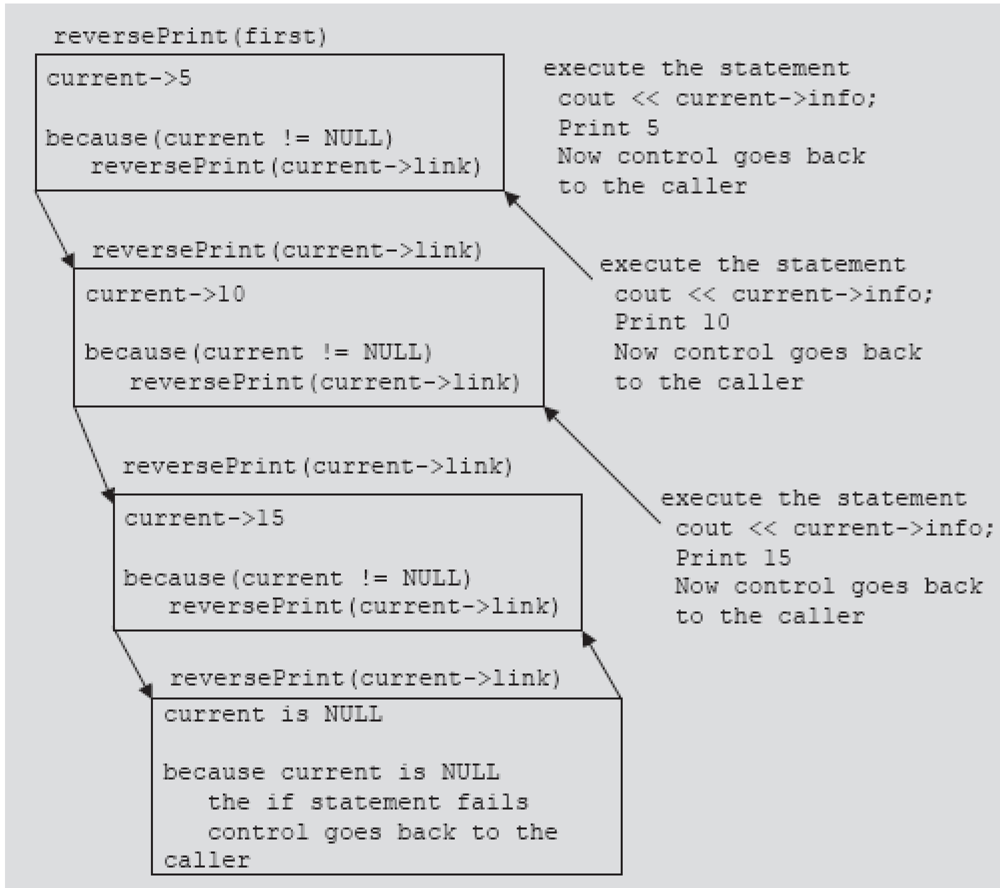

FIGURE 6-6 Execution of the statement `reversePrint(first);`

Print a Linked List in Reverse Order (cont'd.)

- The function `printListReverse`
  - Prints an ordered linked list contained in an object of the type `linkedListType`

```
template <class Type>
void linkedListType<Type>::printListReverse() const
{
    reversePrint(first);
    cout << endl;
}
```

Fibonacci Number

- Sequence: 1, 1, 2, 3, 5, 8, 13, 21, 34 . . .
- Given first two numbers (a_1 and a_2)
  - nth number a_n, $n >= 3$, of sequence given by: $a_n = a_{n-1} + a_{n-2}$
- Recursive function: `rFibNum`
  - Determines desired Fibonacci number
  - Parameters: three numbers representing first two numbers of the Fibonacci sequence and a number n, the desired nth Fibonacci number
  - Returns the nth Fibonacci number in the sequence

Fibonacci Number (cont'd.)

- Third Fibonacci number
  - Sum of first two Fibonacci numbers
- Fourth Fibonacci number in a sequence
  - Sum of second and third Fibonacci numbers
- Calculating fourth Fibonacci number
  - Add second Fibonacci number and third Fibonacci number

Fibonacci Number (cont'd.)

- Recursive algorithm
  - Calculates *n*th Fibonacci number
    - *a* denotes first Fibonacci number
    - *b* denotes second Fibonacci number
    - *n* denotes *n*th Fibonacci number

$$rFibNum(a, b, n) = \begin{cases} a & \text{if } n = 1 \\ b & \text{if } n = 2 \\ rFibNum(a, b, n-1) + \\ rFibNum(a, b, n-2) & \text{if } n > 2. \end{cases} \quad \text{(Equation 6-3)}$$

Fibonacci Number (cont'd.)

- Recursive function implementing algorithm
- Trace code execution
- Review code on page 368 illustrating the function `rFibNum`

```
int rFibNum(int a, int b, int n)
{
    if (n == 1)
        return a;
    else if (n == 2)
        return b;
    else
        return rFibNum(a, b, n - 1) + rFibNum(a, b, n - 2);
}
```

Fibonacci Number (cont'd.)

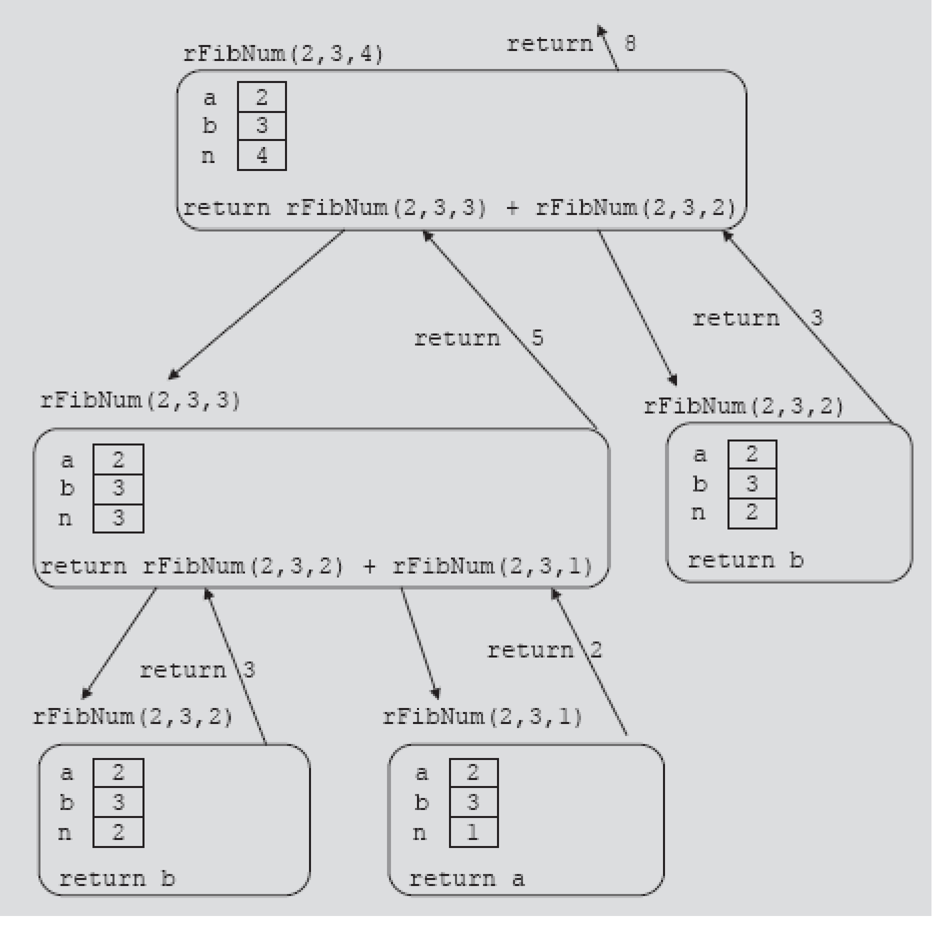

FIGURE 6-7 Execution of `rFibNum(2, 3, 4)`

Tower of Hanoi

- Object
  - Move 64 disks from first needle to third needle
- Rules
  - Only one disk can be moved at a time
  - Removed disk must be placed on one of the needles
  - A larger disk cannot be placed on top of a smaller disk

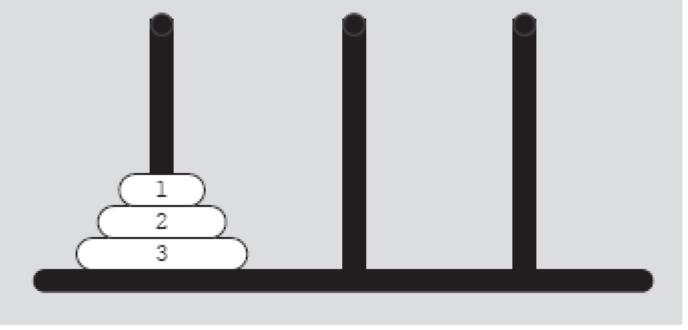

FIGURE 6-8 Tower of Hanoi problem with three disks

Tower of Hanoi (cont'd.)

- Case: first needle contains only one disk
  - Move disk directly from needle 1 to needle 3
- Case: first needle contains only two disks
  - Move first disk from needle 1 to needle 2
  - Move second disk from needle 1 to needle 3
  - Move first disk from needle 2 to needle 3
- Case: first needle contains three disks

Tower of Hanoi (cont'd.)

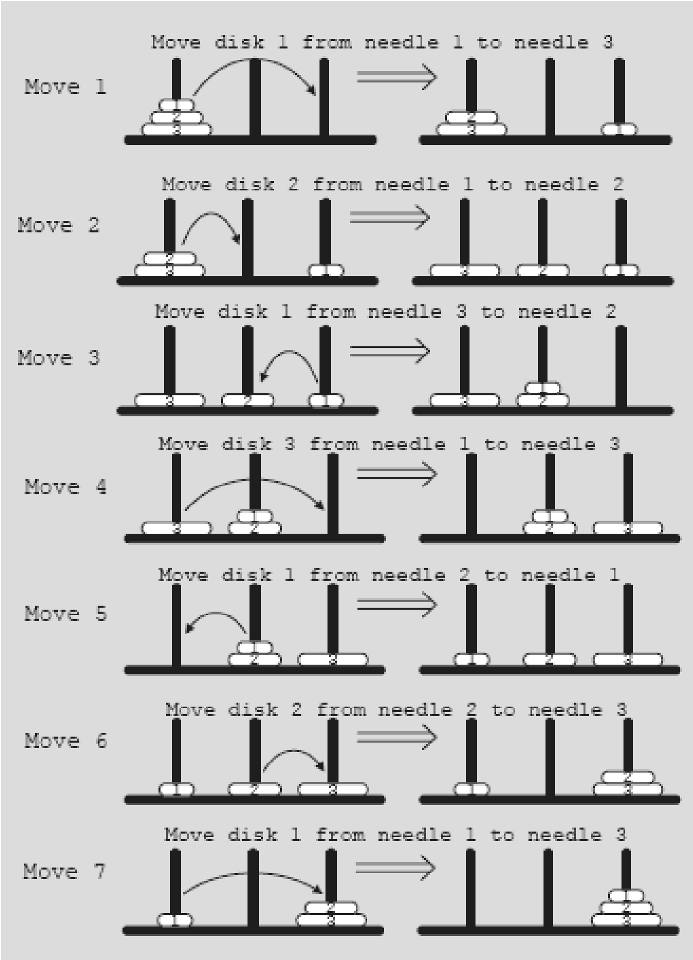

FIGURE 6-9 Solution to Tower of Hanoi problem with three disks

Tower of Hanoi (cont'd.)

- Generalize problem to the case of 64 disks
  - Recursive algorithm in pseudocode

Suppose that needle 1 contains n disks, where $n \geq 1$.

1. Move the top $n - 1$ disks from needle 1 to needle 2, using needle 3 as the intermediate needle.
2. Move disk number n from needle 1 to needle 3.
3. Move the top $n - 1$ disks from needle 2 to needle 3, using needle 1 as the intermediate needle.

Tower of Hanoi (cont'd.)

- Generalize problem to the case of 64 disks
  - Recursive algorithm in C++

```
void moveDisks(int count, int needle1, int needle3, int needle2)
{
    if (count > 0)
    {
        moveDisks(count - 1, needle1, needle2, needle3);

        cout << "Move disk " << count << " from " << needle1
             << " to " << needle3 << "." << endl;

        moveDisks(count - 1, needle2, needle3, needle1);
    }
}
```

Tower of Hanoi (cont'd.)

- Analysis of Tower of Hanoi
  - Time necessary to move all 64 disks from needle 1 to needle 3
  - Manually: roughly 5 x 10^{11} years
    - Universe is about 15 billion years old (1.5 x 10^{10})
  - Computer: 500 years
    - To generate 2^{64} moves at the rate of 1 billion moves per second

Converting a Number from Decimal to Binary

- Convert nonnegative integer in decimal format (base 10) into equivalent binary number (base 2)
- Rightmost bit of *x*
  - Remainder of *x* after division by two
- Recursive algorithm pseudocode
  - Binary(num) denotes binary representation of num

1. `binary(num)` = `num` if `num` = `0`.
2. `binary(num)` = `binary(num / 2)` followed by `num % 2` if `num` > `0`.

Converting a Number from Decimal to Binary (cont'd.)

- Recursive function implementing algorithm

```
void decToBin(int num, int base)
{
    if (num > 0)
    {
        decToBin(num / base, base);
        cout << num % base;
    }
}
```

Converting a Number from Decimal to Binary (cont'd.)

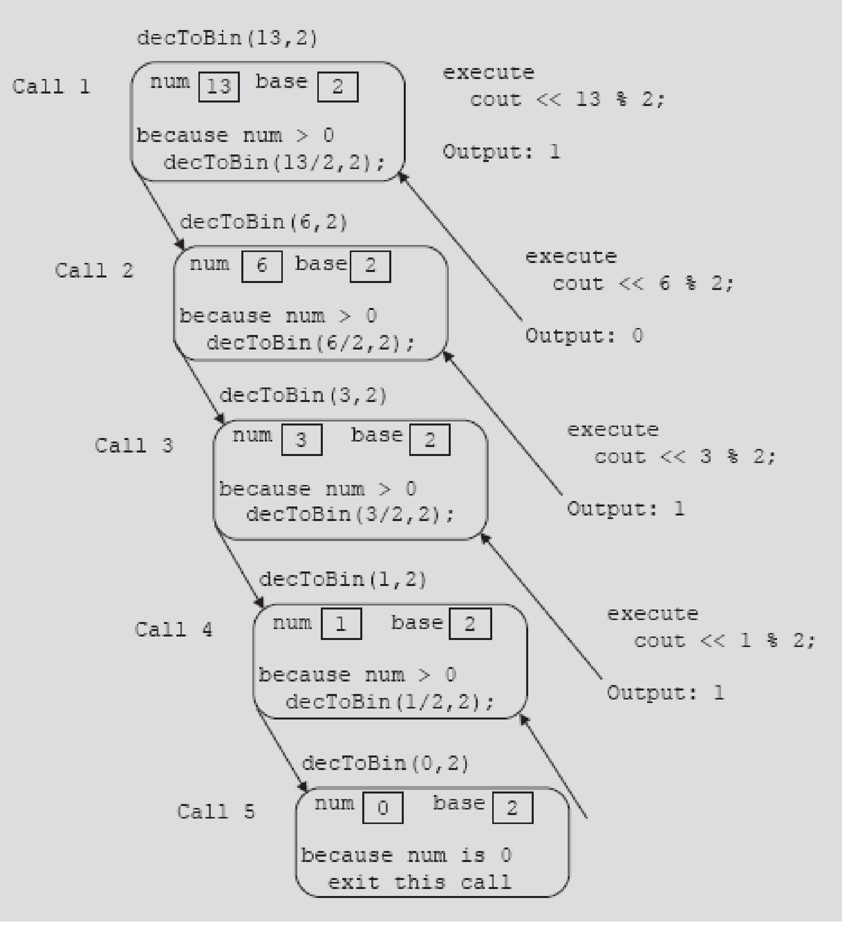

FIGURE 6-10 Execution of `decToBin(13, 2)`

Recursion or Iteration?

- Dependent upon nature of the solution and efficiency
- Efficiency
  - Overhead of recursive function: execution time and memory usage
    - Given speed memory of today's computers, we can depend more on how programmer envisions solution
  - Use of programmer's time
  - Any program that can be written recursively can also be written iteratively

Recursion and Backtracking: 8-Queens Puzzle

- 8-queens puzzle
  - Place 8 queens on a chess-board
    - No two queens can attack each other
  - Nonattacking queens
    - Cannot be in same row, same column, same diagonals

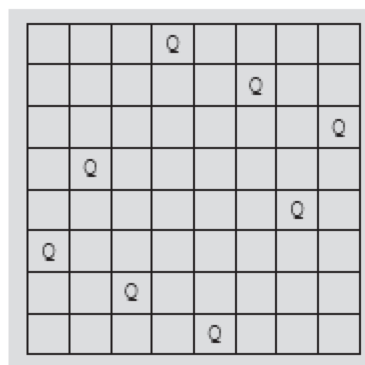

FIGURE 6-11 A solution to the 8-queens puzzle

Recursion and Backtracking: 8-Queens Puzzle (cont'd.)

- Backtracking algorithm
  - Find problem solutions by constructing partial solutions
  - Ensures partial solution does not violate requirements
  - Extends partial solution toward completion
  - If partial solution does not lead to a solution (dead end)
    - Algorithm backs up
    - Removes most recently added part
    - Tries other possibilities

Recursion and Backtracking: 8-Queens Puzzle (cont'd.)

- *n*-Queens Puzzle
  - In backtracking, solution represented as
    - *n*-tuple $(x_1, x_2, . . ., x_n)$
    - Where x_i is an integer such that $1 <= x_i <= n$
    - x_i specifies column number, where to place the *i*th queen in the *i*th row
  - Solution example for Figure 6-11
    - (4,6,8,2,7,1,3,5)
    - Number of 8-tuple representing a solution: 8!

Recursion and Backtracking: 8-Queens Puzzle (cont'd.)

- *n*-Queens Puzzle (cont'd.)
  - 4-queens puzzle

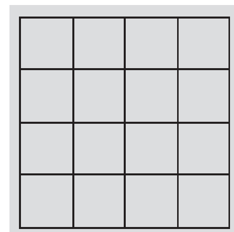

FIGURE 6-12 Square board for the 4-queens puzzle

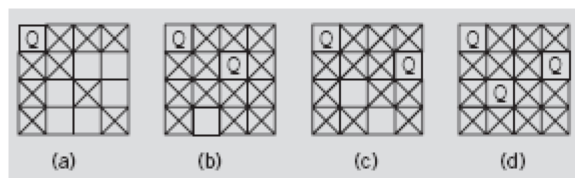

FIGURE 6-13 Finding a solution to the 4-queens puzzle

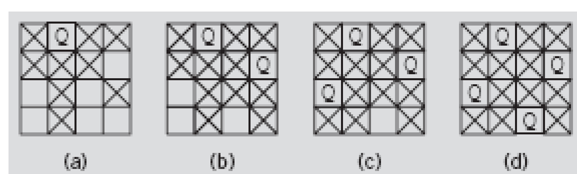

FIGURE 6-14 A solution to the 4-queens puzzle

Recursion and Backtracking: 8-Queens Puzzle (cont'd.)

- Backtracking and the 4-Queens Puzzle
  - Rows and columns numbered zero to three
  - Backtracking algorithm can be represented by a tree

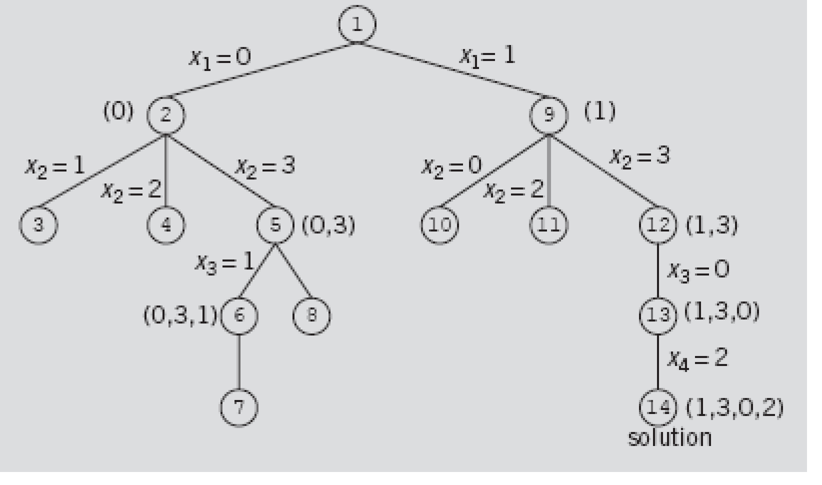

FIGURE 6-15 4-queens tree

Recursion and Backtracking: 8-Queens Puzzle (cont'd.)

- 8-Queens Puzzle
  - Easy to determine whether two queens in same row or column
  - Determine if two queens on same diagonal
    - Given queen at position (i, j), (row i and column j), and another queen at position (k, l) , (row k and column l)
    - Two queens on the same diagonal if $|j – l| = |i – k|$, where $|j – l|$ is the absolute value of $j – l$ and so on
  - Solution represented as an 8-tuple
    - Use the array `queensInRow` of size eight
    - Where `queensInRow[k]` specifies column position of the kth queen in row k

Recursion and Backtracking: 8-Queens Puzzle (cont'd.)

- 8-Queens Puzzle (cont'd.)

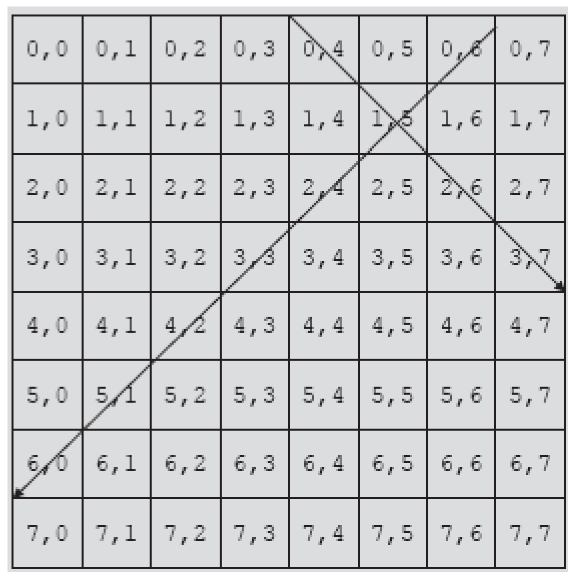

FIGURE 6-16 8 x 8 square board

Recursion and Backtracking: 8-Queens Puzzle (cont'd.)

- 8-Queens Puzzle (cont'd.)
  - General algorithm for the function `canPlaceQueen(k, i)`

```
for (int j = 0; j < k; j++)
    if((queensInRow[j] == i)  //there is already a queen in column i
       || (abs(queensInRow[j] - i) == abs(j-k))) //there is already
                                          //a queen in one of the diagonals
                                          //on which square (k,i) lies
    return false;

return true;
```

Recursion, Backtracking, and Sudoku

- Recursive algorithm
  - Start at first row and find empty slot
  - Find first number to place in this slot
  - Find next empty slot, try to place a number in that slot
  - Backtrack if necessary; place different number
  - No solution if no number can be placed in slot

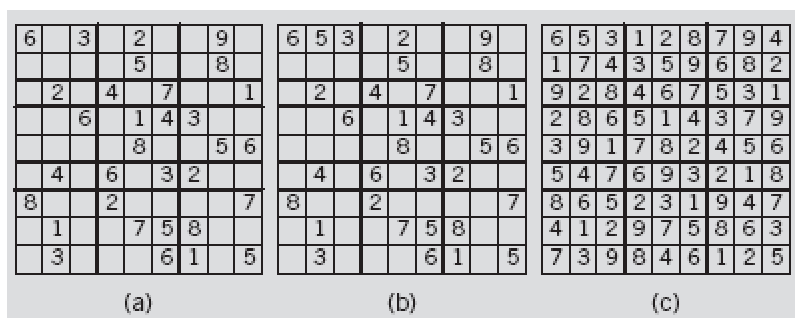

FIGURE 6-17 Sudoku problem and its solution

Recursion, Backtracking, and Sudoku (cont'd.)

- See code on page 384
  - Class implementing Sudoku problem as an ADT
  - General algorithm in pseudocode
    - Find the position of the first empty slot in the partially filled grid
    - If the grid has no empty slots, return true and print the solution
    - Suppose the variables row and col specify the position of the empty grid position

Recursion, Backtracking, and Sudoku (cont'd.)

- General algorithm in pseudocode (cont'd.)

```
for (int digit = 1; digit <= 9; digit++)
{
   if (grid[row][col] <> digit)
   {
       grid[row][col] = digit;
       recursively fill the updated grid;
       if the grid is filled successfully, return true,
       otherwise remove the assigned digit from grid[row][col]
       and try another digit.
    }
  If all the digits have been tried and nothing worked, return false.
```

Recursion, Backtracking, and Sudoku (cont'd.)

- Function definition

```
bool sudoku::solveSudoku()
{
    int row, col;

    if (findEmptyGridSlot(row, col))
    {
        for (int num = 1; num <= 9; num++)
        {
            if (canPlaceNum(row, col, num))
            {
                grid[row][col] = num;
                if (solveSudoku()) //recursive call
                    return true;
                grid[row][col] = 0;
            }
        }

        return false; //backtrack
    }
    else
        return true; //there are no empty slots
}
```

Summary

- Recursion
  - Solve problem by reducing it to smaller versions of itself
- Recursive algorithms implemented using recursive functions
  - Direct, indirect, and infinite recursion
- Many problems solved using recursive algorithms
- Choosing between recursion and iteration
  - Nature of solution; efficiency requirements
- Backtracking
  - Problem solving; iterative design technique