Objectives

- To understand how to think recursively
- To learn how to trace a recursive method
- To learn how to write recursive algorithms and methods for searching arrays

Recursion

- Recursion can solve many programming problems that are difficult to conceptualize and solve linearly
- In the field of artificial intelligence, recursion often is used to write programs that exhibit intelligent behavior:
  - playing games of chess
  - proving mathematical theorems
  - recognizing patterns, and so on
- Recursive algorithms can
  - compute factorials
  - compute a greatest common divisor
  - process data structures (strings, arrays, linked lists, etc.)
  - search efficiently using a binary search

Recursive Thinking

Recursive Thinking

- Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that are difficult to solve by other means
- Recursion reduces a problem into one or more simpler versions of itself

Recursive Thinking (cont.)

Recursive Algorithm to Process Nested Figures

`if` there is one figure

    do whatever is required to the figure

`else`

    do whatever is required to the outer figure

    process the figures nested inside the outer figure in the same way

Recursive Thinking (cont.)

- Consider searching for a target value in an array
  - Assume the array elements are sorted in increasing order
  - We compare the target to the middle element and, if the middle element does not match the target, search either the elements before the middle element or the elements after the middle element
  - Instead of searching *n* elements, we search $n/2$ elements

Recursive Thinking (cont.)

Recursive Algorithm to Search an Array

`if` the array is empty

    return -1 as the search result

`else if` the middle element matches the target

    return the subscript of the middle element as the result

`else if` the target is less than the middle element

    recursively search the array elements preceding the middle element and return the result

`else`

    recursively search the array elements following the middle element and return the result

Steps to Design a Recursive Algorithm

- There must be at least one case (the base case), for a small value of *n*, that can be solved directly
- A problem of a given size *n* can be reduced to one or more smaller versions of the same problem (recursive case(s))
- Identify the base case(s) and solve it/them directly
- Devise a strategy to reduce the problem to smaller versions of itself while making progress toward the base case
- Combine the solutions to the smaller problems to solve the larger problem

Recursive Algorithm for Finding the Length of a String

`if` the string is empty (has no characters)

    the length is 0

`else`

    the length is 1 plus the length of the string that excludes the first character

Recursive Algorithm for Finding the Length of a String (cont.)

```
/** Recursive method length
    @param str The string
    @return The length of the string
*/
public static int length(String str) {
  if (str == null || str.equals(""))
     return 0;
  else
     return 1 + length(str.substring(1));
}
```

Recursive Algorithm for Printing String Characters

```
/** Recursive method printChars
    post: The argument string is
          displayed, one character per line
    @param str The string
*/
public static void printChars(String str) {
  if (str == null || str.equals(""))
    return;
  else {
    System.out.println(str.charAt(0));
    printChars(str.substring(1));
  }
}
```

Recursive Algorithm for Printing String Characters in Reverse

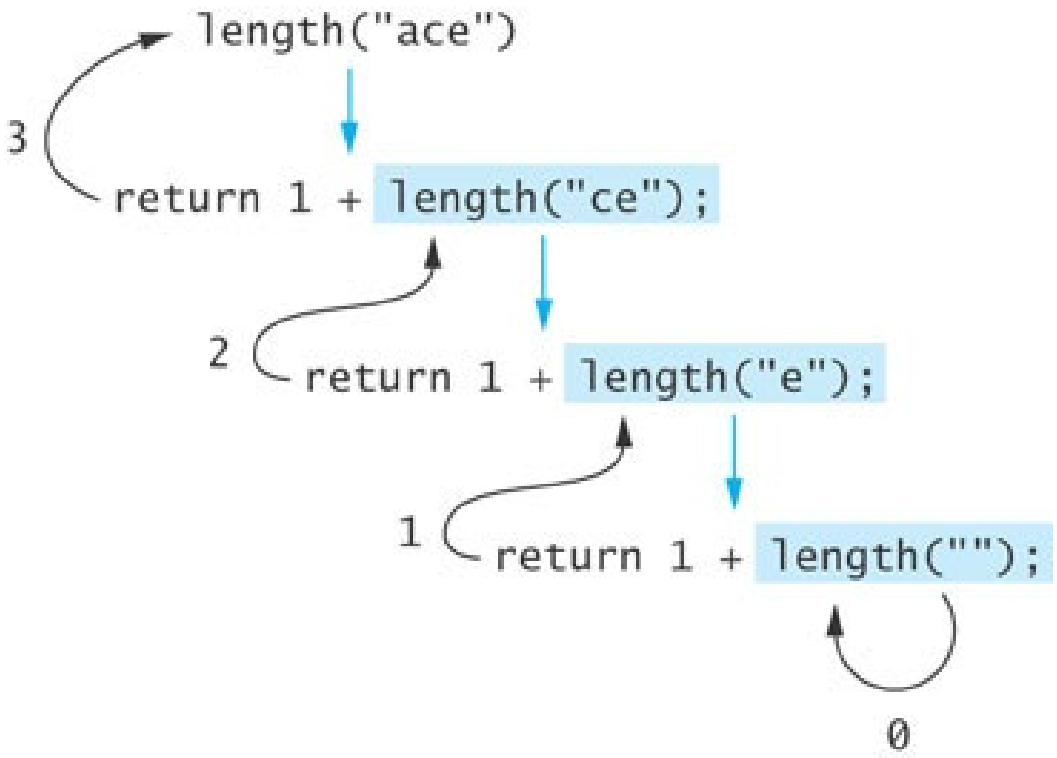
```
/** Recursive method printCharsReverse
    post: The argument string is displayed in reverse,
          one character per line
    @param str The string
*/
public static void printCharsReverse(String str) {
  if (str == null || str.equals(""))
    return;
  else {
    printCharsReverse(str.substring(1));
    System.out.println(str.charAt(0));
  }
}
```

Proving That a Recursive Method Is Correct

- Proof by induction
  - Prove the theorem is true for the base case
  - Show that if the theorem is assumed true for n, then it must be true for n+1
- Recursive proof is similar to induction
  - Verify the base case is recognized and solved correctly
  - Verify that each recursive case makes progress toward the base case
  - Verify that if all smaller problems are solved correctly, then the original problem also is solved correctly

Tracing a Recursive Method

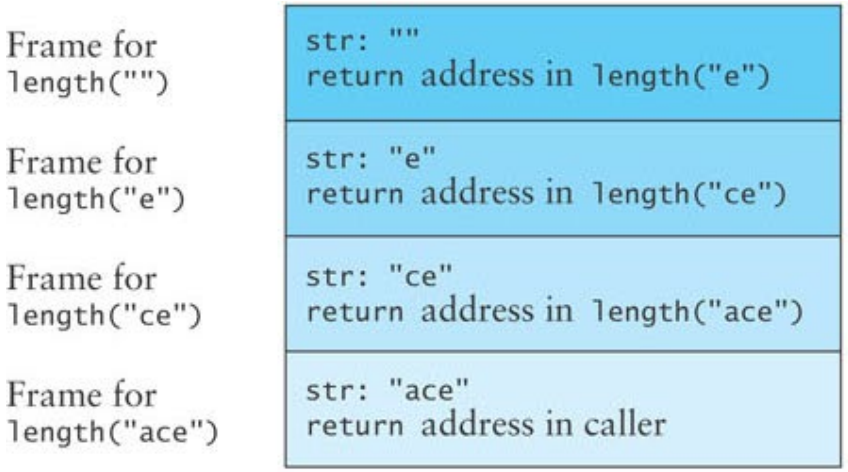
- The process of returning from recursive calls and computing the partial results is called *unwinding the recursion*



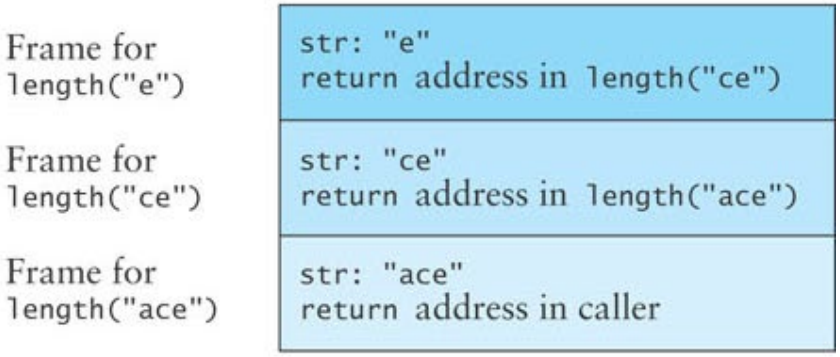
Run-Time Stack and Activation Frames

- Java maintains a run-time stack on which it saves new information in the form of an *activation frame*
- The activation frame contains storage for
  - method arguments
  - local variables (if any)
  - the return address of the instruction that called the method
- Whenever a new method is called (recursive or not), Java pushes a new activation frame onto the run-time stack

Run-Time Stack and Activation Frames (cont.)



Run-time stack after all calls



Run-time stack after return from last call

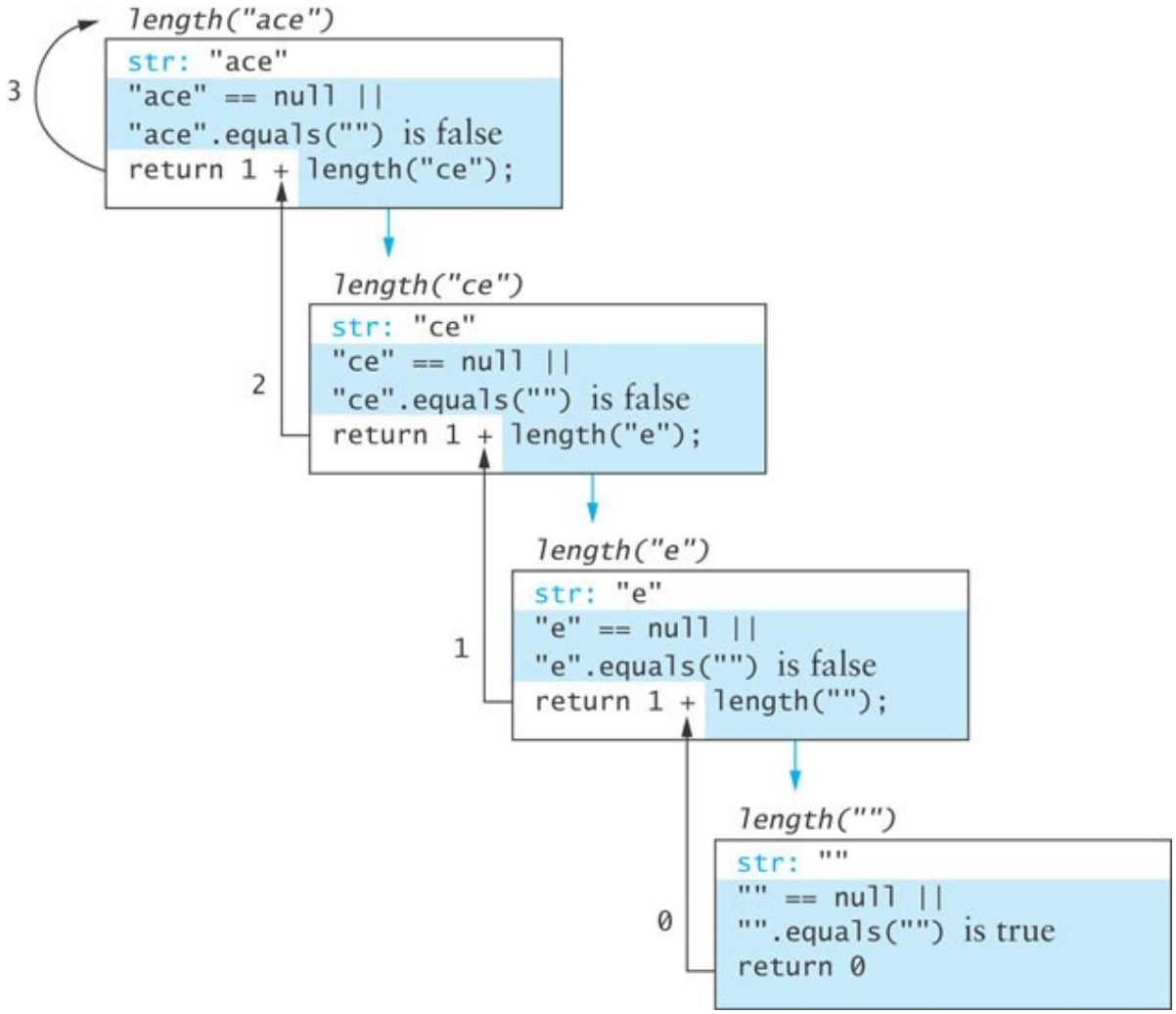
Analogy for the Run-Time Stack for Recursive Calls

- An office tower has an employee on each level each with the same list of instructions
  - The employee on the bottom level carries out part of the instructions, calls the employee on the next level up and is put on hold
    - The employee on the next level completes part of the instructions and calls the employee on the next level up and is put on hold
      - The employee on the next level completes part of the instructions and calls the employee on the next level up and is put on hold
        - The employee on the next level completes part of the instructions and calls the employee on the next level up and is put on hold, and so on until the top level is reached

Analogy for the Run-Time Stack for Recursive Calls (cont.)

- When the employee on the top level finishes the instructions, that employee returns an answer to the employee below
  - The employee below resumes, and when finished, returns an answer to the employee below
    - The employee below resumes, and when finished, returns an answer to the employee below
      - The employee below resumes, and when finished, returns an answer to the employee below, and so on
- Eventually the bottom is reached, and all instructions are executed

Run-Time Stack and Activation Frames



Recursive Definitions of Mathematical Formulas

Recursive Definitions of Mathematical Formulas

- Mathematicians often use recursive definitions of formulas that lead naturally to recursive algorithms
- Examples include:
  - factorials
  - powers

Factorial of *n*: *n*!

- The factorial of *n*, or *n*! is defined as follows:

  $0! = 1$

  $n! = n \times (n-1)! \quad (n > 0)$

- The base case: *n* is equal to 0
- The second formula is a recursive definition

Factorial of *n*: *n*! (cont.)

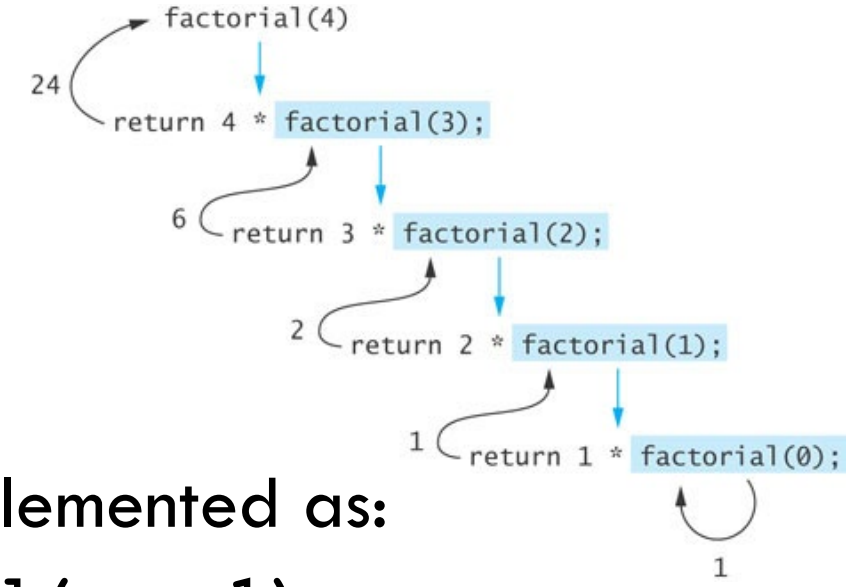
- The recursive definition can be expressed by the following algorithm:

  `if` *n* equals 0

  n! is 1

  `else`

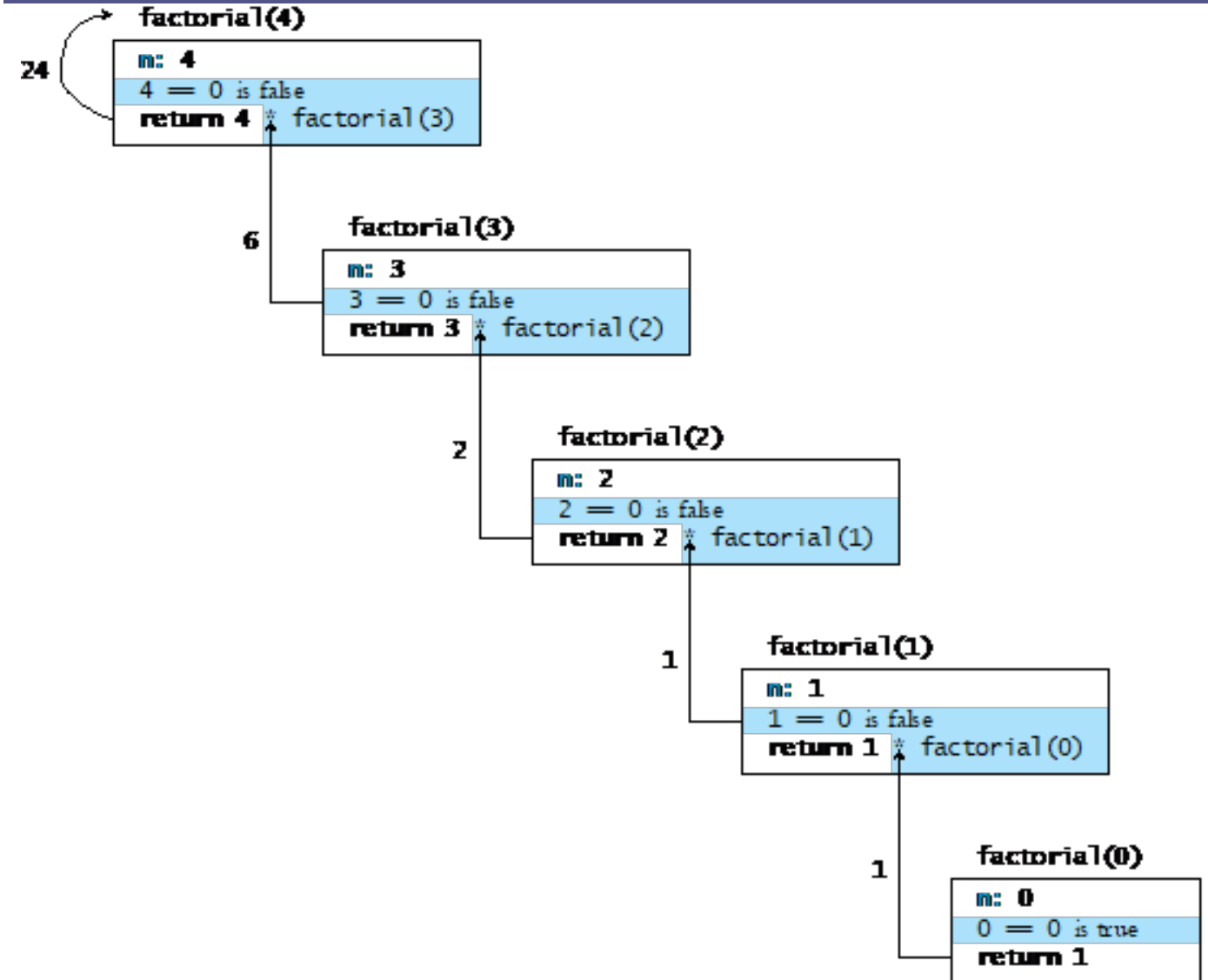
  n! = *n* x (*n* – 1)!



- The last step can be implemented as:

```
return n * factorial(n - 1);
```

Trace factorial(4) w/ activation frames



Infinite Recursion and Stack Overflow

- If you call method `factorial` with a negative argument, the recursion will not terminate because `n` will never equal `0`
- If a program does not terminate, it will eventually throw the `StackOverflowError` exception
- Make sure your recursive methods are constructed so that a stopping case is always reached
- In the `factorial` method, you could throw an `IllegalArgumentException` if `n` is negative

Recursive Algorithm for Calculating x^n

Recursive Algorithm for Calculating x^n ($n \geq 0$)

if *n is 0*
The result is 1
else
The result is $x \times x^{n-1}$

```
/** Recursive power method (in RecursiveMethods.java).
    pre: n >= 0
    @param x The number being raised to a power
    @param n The exponent
    @return x raised to the power n
*/
public static double power(double x, int n) {
    if (n == 0)
 return 1;
    else
 return x * power(x, n - 1);
}
```

Recursion Versus Iteration

- There are similarities between recursion and iteration
- In iteration, a loop repetition condition determines whether to repeat the loop body or exit from the loop
- In recursion, the condition usually tests for a base case
- You can always write an iterative solution to a problem that is solvable by recursion
- A recursive algorithm may be simpler than an iterative algorithm and thus easier to write, code, debug, and read

Iterative `factorial` Method

```
/** Iterative factorial method.
    pre: n >= 0
    @param n The integer whose factorial is being computed
    @return n!
*/
public static int factorialIter(int n) {
    int result = 1;
    for (int k = 1; k <= n; k++)
 result = result * k;
    return result;
}
```

Efficiency of Recursion

- Recursive methods often have slower execution times relative to their iterative counterparts
- The overhead for loop repetition is smaller than the overhead for a method call and return
- If it is easier to conceptualize an algorithm using recursion, then you should code it as a recursive method
- The reduction in efficiency usually does not outweigh the advantage of readable code that is easy to debug