Ch. 3: Recursion

Recursive Solutions

Recursion
- –An extremely powerful problem-solving technique
- –Breaks a problem into smaller identical problems
- –An alternative to iteration, but not always a better one
  - •An iterative solution involves loops

"Recursion can provide elegantly simple solutions to problems of great complexity. However, some recursive solutions are impractical because they are so inefficient."

Recursive Solutions

Four questions for constructing recursive solutions

1. Can you define the problem in terms of a smaller problem of the same type?
2. 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?

Binary Search vs Sequential Search

Some complex and time-consuming problems have recursive solutions that are very simple

A running example that the author uses in Ch. 3 is Binary Search.

- Suppose you are given a sequence of values that are stored in non-decreasing order and you want to locate a particular value in that sequence.

Binary Search

A high-level binary search of an in-order array

```
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)
  } // end if
} // end if
```

Binary search is an example of a "divide-and-conquer" solution

Characteristics of Recursive Methods

1. One of the actions in the method is to call itself, one or more times = a *recursive call*.
2. Each successive recursive call involves an identical, but smaller problem.
3. Recursion ends when the problem size satisfies a condition identifying a single base case or one of a number of base cases.
4. Eventually, a base case is executed and the recursion stops.

Recursive Functions

- The easiest examples of recursion to understand are functions in which the recursion is clear from the definition. As an example, consider the factorial function, which can be defined in either of the following ways:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$$

$$n! = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

- The second definition leads directly to the following code:

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

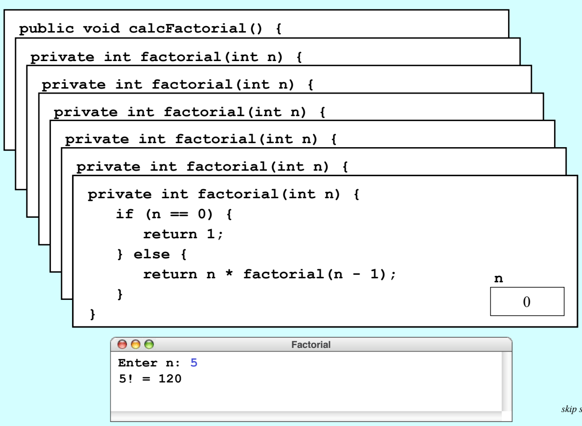
Static Methods

Methods that don't need access to instance variables and are self-contained (depending only on parameter input) are good candidates to be designated as static methods.

All the recursive methods in Ch. 3 of our book are declared static because they only depend on parameter values.

Static methods can be easily tested with DrJava.

Simulating the `factorial` Method



skip simulation

The Recursive "Leap of Faith"

- The purpose of going through the complete decomposition of the calculation of **`factorial(5)`** is to convince you that the process works and that recursive calls are in fact no different from other method calls, at least in their internal operation.

  Our book uses a systematic trace of recursive methods called a **box trace**, very similar to the method stack shown on the last slide.

- As you write a recursive program, it is important to believe that any recursive call will return the correct answer as long as the arguments continually get closer to a stopping condition.
- Believing that to be true—even before you have completed the code—is called the **recursive leap of faith**.

The Recursive Paradigm

- Most recursive methods you encounter in an introductory course have bodies that fit the following general pattern:

```
if (test for a simple case) {
   Compute and return the simple solution without using recursion.
} else {
   Divide the problem into one or more subproblems that have the same form.
   Solve each of the subproblems by calling this method recursively.
   Return the solution from the results of the various subproblems.
}
```

- Finding a recursive solution is mostly a matter of figuring out how to break it down so that it fits the paradigm. When you do so, you must do two things:
  1. Identify **simple cases** that can be solved without recursion.
  2. Find a **recursive decomposition** that breaks each instance of the problem into simpler subproblems of the same type, which you can then solve by applying the method recursively.

Tracing Recursive Methods

Box trace

- A systematic way to trace the actions of a recursive method
- Each box roughly corresponds to an activation record
- An activation record
  - Contains a method's local environment at the time of and as a result of the call to the method

Box Trace

1. Label each recursive call in the body of the recursive method.
2. Represent each call to the method by a new box containing the method's local environment.
   a) values of local variables and parameters
   b) placeholder for return value and operation performed
3. Draw an arrow from box to box, where each represents another recursive call.
4. Cross off boxes as methods return

Tracing the fact method

- A method's local environment includes:
  - The method's local variables
  - A copy of the actual value arguments
  - A return address in the calling routine
  - The value of the method itself

```
n = 3
A: fact(n-1) = ?
return 3 * ?
```

Recursive Methods

As a programmer, you need to ensure that all recursive calls bring the execution closer to the stopping condition.

The simpler cases must eventually reach the stopping condition or the method will call itself infinitely.

In Java, when a method calls itself a very large number of times, the stack gets full and a "Stack Overflow" occurs.

A Recursive `void` Method: Writing a String Backward

Problem

Given a string of characters, print it in reverse order

Recursive solution

Each recursive step of the solution diminishes by 1 the length of the string to be written backward

Base case

Print the empty string backward

Iterative Version

- This is an iterative method to print a String in reverse

```
public static void writeStringBackwards(String s) {

     for (int i = s.length()-1; i >= 0; i--) {
        System.out.print(s.charAt(i));
     }
     System.out.println();
  }
```

Recursive Version

- This is a recursive method to print a String in reverse

```
public static void writeBackward(String s, int size) {
     if (size==0) {
        System.out.println();
     }
     else {
        // print the last character
        System.out.print(s.substring(size-1, size));
        // write the rest of the string in reverse
        writeBackward(s, size - 1);
     }
  }
```

In this example, the base case is reached when size = 0.

The recursive call is made on the input string minus the *last* character (str length is closer to the empty string).

Recursive Version

- This is another recursive method to print a String in reverse

```
public static void writeBackward(String s) {
   if (s.length() > 0) {
      // write the rest of the string backward
      writeBackward(s.substring(1));
      // print the first character
      System.out.print(s.charAt(0));
   }
   System.out.println();
}
```

Like the last example, the base case is reached when size = 0.

The recursive call is made on the input string minus the *first* character (str length is closer to the empty string).

Recursive Methods

- This is a recursive method to reverse a String, returning a String instead of printing one out.

```
public static String recRevString(String str) {
   if (str.length() == 0) {
      return "";
   } else {
      return recRevString(str.substring(1)) +
              str.charAt(0);
   }
}
```

In this example, the base case is reached when the length of the input string is zero.
The recursive call is made on the input string minus the first character (str length is closer to the empty string).

Recursive Methods

- Exercise: Write a recursive method to return the sum of all the numbers between 1 and n

Recursive Methods

- Exercise: Write a recursive method to return the sum of all the numbers in a given input array

Recursive Methods

- Exercise: Write a recursive method to raise a base to an exponent power

These examples illustrate the essential features of a recursive method:

1. A base case that has no recursive call.
2. A recursive case that contains a call to the containing method, passing in an argument that is closer to the base case than the value of the current parameter.

Recursive Methods

- Exercise: Write a recursive method to determine if a given String is a palindrome.

Next three problems

– Require you to count certain events or combinations of events or things
– Contain more than one base cases
– Are good examples of inefficient recursive solutions

Multiplying Rabbits (The Fibonacci Sequence)

- "Facts" about rabbits
  - Rabbits never die
  - A rabbit reaches sexual maturity exactly two months after birth, that is, at the beginning of its third month of life
  - Rabbits are always born in male-female pairs
    - At the beginning of every month, each sexually mature male-female pair gives birth to exactly one male-female pair

Multiplying Rabbits (The Fibonacci Sequence)

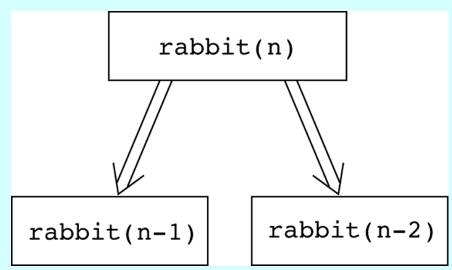
Problem

How many pairs of rabbits are alive in month n?

Recurrence relation

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

Multiplying Rabbits (The Fibonacci Sequence)



Multiplying Rabbits (The Fibonacci Sequence)

Base cases

rabbit(2), rabbit(1)

Recursive definition

$$\text{rabbit}(n) = \begin{cases} 1 & \text{if } n \text{ is 1 or 2} \\ \text{rabbit}(n-1) + \text{rabbit}(n-2) & \text{if } n > 2 \end{cases}$$

Fibonacci sequence

The series of numbers rabbit(1), rabbit(2), rabbit(3), and so on

NOT an efficient solution for this problem because each solution requires many redundant computations

Organizing a Parade

Rules about organizing a parade

The parade will consist of bands and floats in a single line
One band cannot be placed immediately after another

Problem

How many ways can you organize a parade of length n?

Organizing a Parade

Let:

- P(n) be the number of ways to organize a parade of length n
- F(n) be the number of parades of length n that end with a float
- B(n) be the number of parades of length n that end with a band

Then

- P(n) = F(n) + B(n)

Finding the largest item in an array

if (array has only one item)

- max(array) is the item

else

- max(array) is the maximum of max(left half of array) and max(right half of array)

Mr. Spock's Dilemma (Choosing k out of n Things)

Problem

- How many different choices are possible for exploring k planets out of n planets in a solar system?

Let

- c(n, k) be the number of groups of k planets chosen from n

Mr. Spock's Dilemma (Choosing k out of n Things)

In terms of Planet X:

c(n, k) = (the number of groups of k planets that include Planet X)

+

(the number of groups of k planets that do not include Planet X)

Mr. Spock's Dilemma (Choosing k out of n Things)

The number of ways to choose k out of n things is the sum of

The number of ways to choose k-1 out of n-1 things
and
The number of ways to choose k out of n-1 things

c(n, k) = c(n-1, k-1) + c(n-1, k)

The End