Data Structures for Java

William H. Ford
William R. Topp

Chapter 6
Recursion

Bret Ford

The Concept of Recursion

- An algorithm is recursive if it can be broken into smaller problems of the same type. Continue this decomposition until one or more of the smaller problems has a solution.
- Backing up a directory and its subdirectories is recursive. At each step, the backup copies files in the current directory and then repeats the process in a subdirectory. Eventually, the backup reaches a directory with no subdirectories.

Iterative factorial()

- n-factorial is the product of the first n integers.

```
n! = n * (n-1) * (n-2) * ... * 2 * 1
```

- n! can be computed iteratively.

```
// n is a non-negative integer
public static int factorial(int n)
{
   int factValue = 1;
   while (n >= 1)
   {
      factValue *= n;
      n--;
   }
   return factValue;
}
```

Recursive Computation of n!

```
Define 0! = 1

4! = 4 * 3!          // multiplication awaits evaluation of 3!
3! = 3 * 2!          // need value of 2!
2! = 2 * 1!          // need value of 1!
1! = 1 * 0!          // need value of 0!
0! = 1               // know 0! = 1 (stopping condition)

1! = 1 * 0!          // compute 1! with value of 0!
   = 1
2! = 2 * 1!          // compute 2! with value of 1!
   = 2 * 1 = 2
3! = 3 * 2!          // compute 3! with value of 2!
   = 3 * 2 = 6
4! = 4 * 3!          // compute 4! with value of 3!
   = 4 * 6 = 24
```

Recursive Computation of n! (continued)

- The computation of n! can be viewed recursively as follows:

```
n! = n * (n-1)!
(n-1)! = (n-1) * (n-2)!
...
2! = 2 * 1
1! = 1 * 0!
0! = 1              (stopping condition)

After reaching the stopping condition, revisit the
multiplication steps in reverse order.
```

Describing a Recursive Algorithm

- The design of a recursive method consists of the following elements:
  - One or more *stopping conditions* that can be directly evaluated for certain arguments.
  - One or more *recursive steps*, in which a current value of the method can be computed by repeated calling of the method with arguments that will eventually arrive at a stopping condition.

Implementing Recursive Methods

- Use an if-else statement to distinguish between a stopping condition and a recursive step.

```
// n is a non-negative integer
public static double factorial(int n)
{
   if (n == 0)
      // stopping condition
      return 1;
   else
      // recursive step
      return n * factorial(n-1);
}
```

Recursive Computation of 1 + 2 + … + n

Example: `S(5) = 5 + (4 + 3 + 2 + 1) = 5 + S(4)`

$$S(n) = \begin{cases} 1 & n = 1 \quad \text{stopping condition} \\ S(n-1) + n & n > 1 \quad \text{recursive step} \end{cases}$$

```
public static int sumToN(int n)
{
   if (n == 1)
      return 1;
   else
      return sumToN(n-1) + n;
}
```

Infinite Recursion

- A recursive algorithm must lead to a stopping condition. Infinite recursion occurs when a stopping condition is never reached.
  - Example: $f(n) = \begin{cases} 0, & n = 0 \\ f(n/4+1)+n, & n \geq 1 \end{cases}$

```
f(5) = f(5/4 + 1) + 5;     // recursive call to f(5/4 + 1) = f(2)
f(2) = f(2/4 + 1) + 2;     // recursive call to f(2/4 + 1) = f(1)
f(1) = f(1/4 + 1) + 1;     // recursive call f(1/4 + 1) = f(1)
     . . .                 // infinite recursion!
```

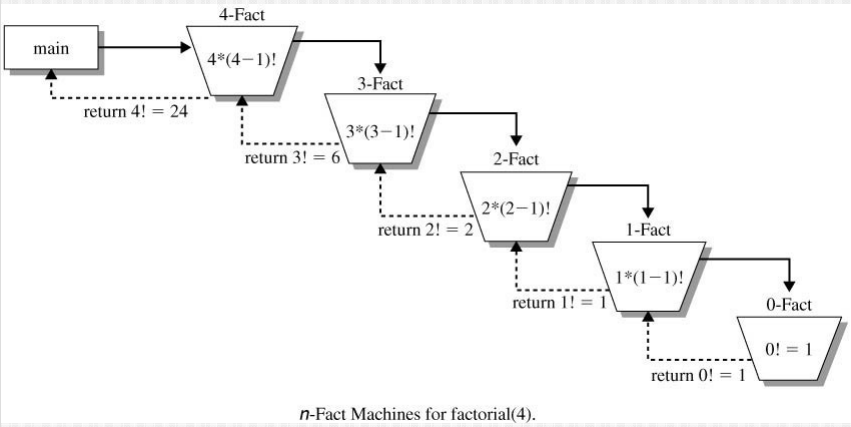
How Recursion Works

- Think of factorial(n) as a machine called n-Fact that computes n! by carrying out the multiplication n*(n-1)!. The machine must be networked with a series of other n-Fact machines that pass information back and forth.
- The 0-Fact machine is an exception and can work independently and produce the result 1 without assistance from another machine.

The n-Fact Machine

```
4-Fact  Starts up the 3-Fact machine
          Waits for return value (3!) from 3-Fact.
          Computes 4 * 3! = 24 and returns to main
3-Fact  Starts up the 2-Fact machine
          Waits for return value (2!) from 2-Fact.
          Computes 3 * 2! = 6 and returns to 4-Fact
2-Fact   Starts up the 1-Fact machine
          Waits for return value (1!) from 1-Fact.
          Computes 2 * 1! = 2 and returns to 3-Fact
1-Fact  Starts up the 0-Fact machine
          Waits for return value (1!) from 1-Fact.
          Computes 1 * 0! = 1 and returns to 2-Fact
0-Fact  Determines that 0! = 1
          Returns 1 to 1-Fact
```

The n-Fact Machine (concluded)



n-Fact Machines for factorial(4).

Multibase Representations

- Decimal is only one representation for numbers. Other bases include 2 (binary), 8 (octal), and 16 (hexadecimal).
  - Hexadecimal uses the digits 0-9 and a=10, b=11, c=12, d=13, e=14, f=16.

$95 = 1011111_2$ // 95 $= 1(2^6)+0(2^5)+0(2^4)+0(2^3)+1(2^2)+1(2^1)+1(2^0)$
$= 1(64)+0(32)+1(16)+1(8)+1(4)+1(2)+1$

$95 = 340_5$ // 95 $= 3(52) + 4(51) + 0(5^0)$
$= 3(25) + 4(5) + 0$

$95 = 137_8$ // 95 $= 1(82) + 3(81) + 7(8^0)$
$= 1(64) + 3(8) + 7$

$748 = 2ec_{16}$ // 748 $= 2(16^2) + 14(16^1) + 12(16^0)$
$= 2(256) + 14(16) + 12$
$= 512 + 224 + 12$

Multibase Representations (2)

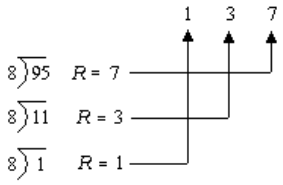
- An integer n > 0 can be represented in different bases using repeated division.
  - Generate the digits of n from right to left using operators '%' and '/'. The remainder is the next digit and the quotient identifies the remaining digits.

Multibase Representations (3)

Step 1: $8\overline{)95}$ quotient 11, remainder 7 (88)

Step 2: $8\overline{)11}$ quotient 1, remainder 3 (8)

Step 3: $8\overline{)1}$ quotient 0, remainder 1 (0)




```
85 % 8 = 7        // remaining digits: 85/8 = 11
11 % 8 = 3        // remaining digits: 11/8 = 1
 1 % 8 = 1        // remaining digits: 1/8 = 0
   0              // stopping condition
```

Multibase Representations (4)

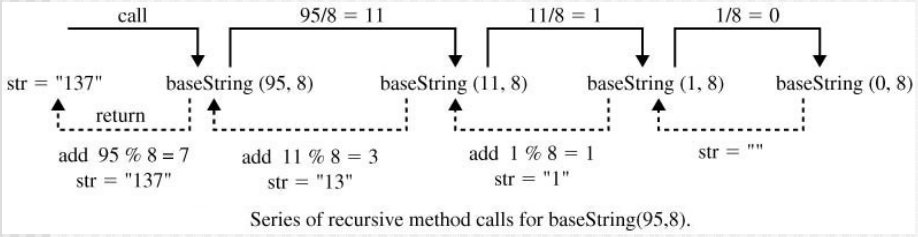
- Convert n to base b by converting the smaller number n/b to base b (recursive step) and adding the digit n%b.

MultibaseRepresentations (5)

```
// returns string representation
// of n as a base b number
public static String baseString(int n, int b)
{
   String str = "", digitChar = "0123456789abcdef";

   // if n is 0, return empty string
   if (n == 0)
      return "";
   else
   {
      // get string for digits in n/b
      str = baseString(n/b, b);   // recursive step
      // return str with next digit appended
      return str + digitChar.charAt(n % b);
   }
}
```

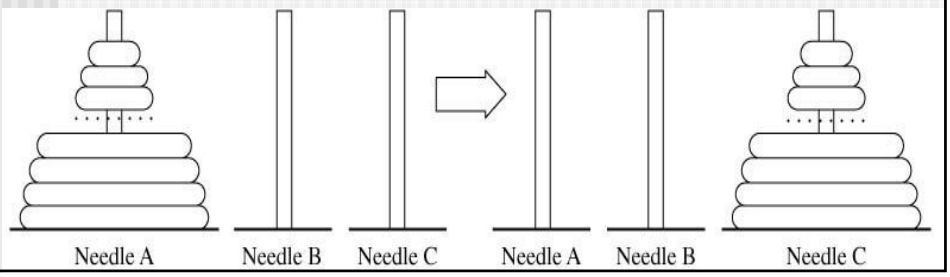
Multibase Representations (6)



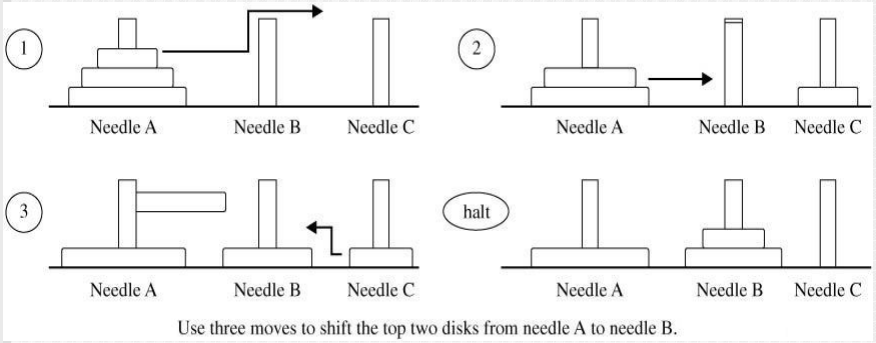
Series of recursive method calls for baseString(95,8).

Towers of Hanoi

- You are given a stack of n graduated disks and a set of three needles called A, B, and C. The initial setup places the n disks on needle A. The task is to move the disks one at a time from needle to needle until the process rebuilds the original stack, but on needle C. In moving a disk, a larger disk may never be placed on top of a smaller disk.

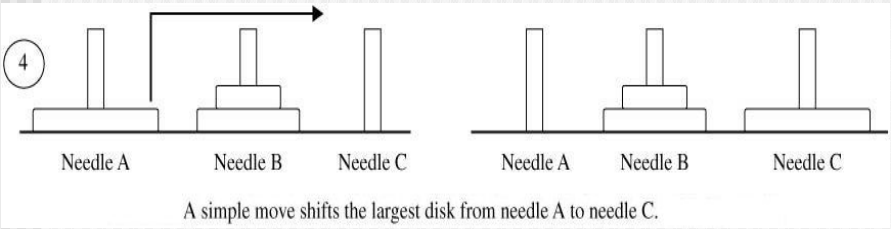


Towers of Hanoi Example 1



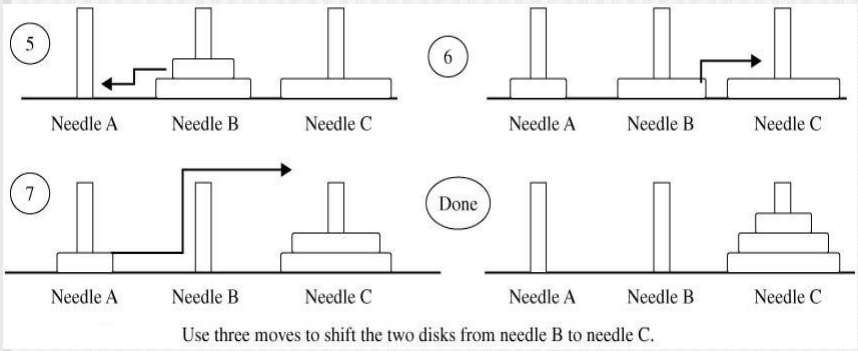
Use three moves to shift the top two disks from needle A to needle B.

Towers of Hanoi Example 2



A simple move shifts the largest disk from needle A to needle C.

Towers of Hanoi Example 3



Use three moves to shift the two disks from needle B to needle C.

Method hanoi()

- The method hanoi() has arguments n, the number of disks, and three string arguments that denote the name of the starting needle (initNeedle), the destination needle (endNeedle), and the intermediate needle (tempNeedle) that temporarily holds disks during the moves.

```
// move n disks from initNeedle to endNeedle
// using tempNeedle for temporary storage
public static void hanoi(int n, String initNeedle,
                         String endNeedle, String tempNeedle)
```

Method hanoi() (continued)

- Move n-1 disks from initNeedle to tempNeedle using endNeedle for temporary storage.

```
hanoi(n-1, initNeedle, tempNeedle, endNeedle);
```

- Move largest disk to endNeedle.

```
System.out.println("Move " + initNeedle + " to " +
                   endNeedle);
```

- Move n-1 disks from tempNeedle to endNeedle using initNeedle for temporary storage.

```
hanoi(n-1, tempNeedle, endNeedle, initNeedle);
```

Method hanoi() (continued)

```
// move n disks from initNeedle to endNeedle, using tempNeedle
// for intermediate storage of the disks
public static void hanoi(int n, String initNeedle,
                         String endNeedle,
                         String tempNeedle)
{
   // stopping condition: move one disk
   if (n == 1)
      System.out.println("move " + initNeedle +
                         " to " + endNeedle);
   else
   {
      // move n-1 disks from initNeedle to
      // tempNeedle using endNeedle
      // for temporary storage
      hanoi(n-1,initNeedle,tempNeedle,endNeedle);
```

Method hanoi() (concluded)

```
         // move largest disk to endNeedle
         System.out.println("move " + initNeedle +
                            " to " + endNeedle);

         // move n-1 disks from tempNeedle to
         // endNeedle using initNeedle
         // for temporary storage
         hanoi(n-1,tempNeedle,endNeedle,initNeedle);
      }
   }
```

Program 6.2

```
import java.util.Scanner;

public class Program6_2
{
   public static void main(String[] args)
   {
      // number of disks and the needle names
      int n;
      String beginNeedle = "A",
             middleNeedle = "B",
             endNeedle = "C";
      // the keyboard input stream
      Scanner keyIn = new Scanner(System.in);
      // prompt for n and solve
      // the puzzle for n disks
      System.out.print("Enter the number of disks: ");
      n = keyIn.nextInt();
```

Program 6.2 (concluded)

```
      System.out.println("The solution for n = " + n);
      hanoi(n, beginNeedle, endNeedle, middleNeedle);
   }

   < method hanoi() listed in the program discussion >
}
```

```
Run:

Enter the number of disks: 3
The solution for n = 3
Move A to C
Move A to B
Move C to B
Move A to C
Move B to A
Move B to C
Move A to C
```

Evaluating Recursion

- Sometimes recursion simplifies algorithm design, but it is sometimes not efficient and an iterative algorithm is preferable.
  - For the Towers of Hanoi, a recursive solution is elegant and easier to code than the corresponding iterative solution.
  - For some problems, a recursive solution is inefficient.

Fibonacci Numbers

- Fibonacci numbers are the sequence of integers beginning with position n = 0. The first two terms are 0 and 1 by definition. Each subsequent term, beginning at n = 2, is the sum of the two previous terms.

n	Value	Sum
2	1	0+1
3	2	1+1
4	3	1+2
5	5	2+3
6	8	3+5

Recursive method fib()

$$fib(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ fib(n-1) + fib(n-2), & n \geq 2 \end{cases}$$

```
// compute Fibonacci number n using recursion
public static int fib(int n)
{
   if (n <= 1)   // stopping conditions
      return n;
   else
      return fib(n-1) + fib(n-2);   // recursive step
}
```

Recursive method fib()
(continued)

- The recursive method fib() makes multiple calls to itself with the same argument. This creates enounous redundancy. If numCall(n) is the number of recursive calls required to evaluate fib(n), then it can be shown that
  numCall(n) = 2 * fib(n+1) - 1
  For example, numCall(35) = 29,860,703
  Because fib(n) gets large quickly, recursion is not an efficient way to compute the Fibonacci numbers.

Iterative method fibIter()

```
// compute Fibonacci number n iteratively
public static int fibIter(int n)
{
   int oneback = 1, twoback = 0, current = 0;
   int i;
   // return is immediate for first two numbers
   if (n <= 1)
      current = n;
   else
      for (i = 2; i <= n; i++)
      {
         current = oneback + twoback;
         // update for next calculation
         twoback = oneback;
         oneback = current;
      }
   return current;
}
```

Program 6.3

```
import ds.time.Timing;

public class Program6_3
{
   public static void main(String[] args)
   {
      int fib_45;
      Timing timer = new Timing();
      double fibTime;
      // evaluate fibIter(45) using iterative method
      System.out.println("Value of fibIter(45) by " +
                         "iteration is " + fibIter(45));
      // evaluate fib(45) using recursive method
      System.out.print("Value of fib(45) by recursion is ");
      // start/stop timing the recursive method
      timer.start();
      fib_45 = fib(45);
      fibTime = timer.stop();
```

Program 6.3 (concluded)

```
        // output the value for fib(45) and time of computation
        System.out.println(fib_45);
        System.out.println(
           "   Time required by the recursive version is " +
           fibTime + " sec");
    }
    < recursive method fib() defined in the
      previous discussion >
    < iterative method fibIter() defined in the
      previous discussion >
}

Run:
Value of fibIter(45) by iteration is 1134903170
Value of fib(45) by recursion is 1134903170
   Time required by the recursive version is 34.719 sec
```

Criteria of Using Recursion

- With the overhead of method calls, a simple recursive method could significantly deteriorate runtime performance. In the case of the Fibonacci numbers, use the O(n) iterative solution in preference to the recursive version.
- Use recursion when it enhances the algorithm design and provides a method implementation that runs with reasonable space and time efficiency.