Implementing Recursion

Always via a method – what makes it recursive is calling itself.

Body always contains an 'if' statement to handle base case(s) plus the recursive case(s).

The recursive call works exactly like calling any other method.

- pass input to caller via parameters
- pass output back from call via return value

Local variables in the method are not accessible elsewhere, even from another call of the same method.

Instance variables are shared by calls of recursive methods, but this can lead to confusion – avoid having “global” state.

Self Test

- factorial

  $n! = n\ (n-1)!$

  $1! = 1$

- fibonacci

  $fib(n) = fib(n-1)+fib(n-2)$

  $fib(2) = 1$

  $fib(1) = 1$

- exponentiation

  $a^0 = 1$

  $a^n = (a^{n/2})(a^{n/2})$ if n is even

  $a^n = a(a^{n-1})$ if n is odd

- implement each of these definitions as a Java function which returns the desired value

The “Magic” of Recursion

Recursion can look like magic – when is all the work getting done?

- It is spread out, not non-existent!

```
public static int fib ( int n ) {
  if ( n == 1 || n == 2 ) { return 1; }
  else { return fib(n-1)+fib(n-2); }
}
```


The “Magic” of Recursion

Correctness is established by induction.

- explain why the base case answer is correct
- assume that the smaller problems are solved correctly, and explain why the larger problem is thus solved correctly

Only need to consider one level at a time!

```
public static int fib ( int n ) {
  if ( n == 1 || n == 2 ) { return 1; }
  else { return fib(n-1)+fib(n-2); }
}
```

base cases are correct: the sequence starts with 1 1

recursive case is correct: by definition, the next fibonacci number is the sum of the previous two

Another Pattern

- for any method, parameters are used to pass the method values it needs to work with
- if the friends need more information than the original call, create a private helper method with the necessary parameters to handle the recursion

Another Pattern

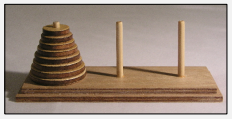
```
/**
 * Print the tree, indenting to show the depth of each element.
 * @param root the root of the subtree to print
 */
public void print ( TreeNode root ) {
  print(root,"");
}

/**
 * Print the subtree rooted at the specified node, indenting to show
 * the depth of each element.
 * @param node root of the subtree to print
 * @param indent amount of indentation for this level of the tree
 */
private void print ( TreeNode root, String indent ) {
  if ( root.getLeft() == null ) {   // leaf
    System.out.println(indent+root.getElement());
  } else {
    System.out.println(indent+root.getElement());
    print(root.getLeft(),indent+" ");
    print(root.getRight(),indent+" ");
  }
}
```

the amount to indent this level – difficult for this method to know for itself, but easy for the caller to pass

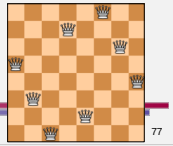
Recursion as a Technique

Recursion is also at the core of two powerful problem-solving strategies –

- divide-and-conquer
  - utilize abstraction – have one or more friends solve smaller versions of the problem for you, and you use those answers to solve your problem
  - examples
    - e.g. towers of hanoi
    - e.g. quicksort

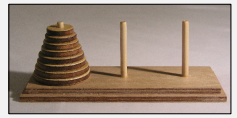


- recursive backtracking
  - leverage tree traversal to enumerate all possible combinations of a series of choices without having to deal with the explosion-of-fingers problem
  - examples
    - e.g. n queens
    - e.g. making change



Divide-and-Conquer

- divide-and-conquer
  - break the task up into one or more smaller instances of the same task
  - have friends solve the smaller instances
  - use those solutions to solve the original problem
- towers of hanoi
  - move *n* disks from one peg to another
  - only one disk can be moved at a time
  - a larger disk cannot be stacked on top of a smaller disk
  - solution – move n disks from peg a to peg b using peg c as a spare
    - move n-1 disks from peg a to peg c (using peg b as a spare)
    - move nth disk from peg a to peg b
    - move n-1 disks from peg c to peg b (using peg a as a spare)



recursive calls

Divide-and-Conquer

- sorting
  - arrange *n* elements in increasing (or decreasing) order
  - mergesort

recursive calls →
    - split elements into two groups of size n/2
    - sort each half
    - merge two halves

  - quicksort
    - pick pivot element

recursive calls →
    - split into smaller and larger groups (based on the pivot)
    - sort each group