Introduction to Recursion

"Recursion, n : see 'Recursion'"

What is recursion?

- Recursion is an approach to problem solving
  - It is typically used to solve problems that break down into smaller (simpler!) versions of themselves
  - Recursive methods are ones that call themselves

A bad example of recursion

- Sample code:

```
public class BadRecursion {
   public void doSomething() {
         System.out.println( "Hi there!" );
         doSomething();
   }
   static void main( String [] args ) {
         (new BadRecursion).doSomething();
   }
}
```

- What will this code do?

Steps for recursion

- There are 4 main steps in solving problems recursively:
  1) Define a test to decide if you should stop (or continue) recursing
  2) Define one or more end cases to terminate the recursion
  3) Define recursive call(s) that continue the function's recursion
  4) Identify any "error cases" that need to be handled.

A classic recursion example

- Consider the "factorial" function
  - n! = n * (n-1) * (n-2) * … * 2 * 1
  - 0! = 1
- An example:
  - 5! = 5 * 4 * 3 * 2 * 1 = 120

Designing factorial recursively

- Design decisions:
  - What is the base case for factorial?
  - Assume that "n" is not the base case, and fill in the blank with a recursive definition of n!
    n! = _____
  - What is the test going to be that decides when to stop recursing?
  - Are there any error cases to be handled?

Writing factorial recursively

- Fill in the code for the method:

```
public long factorial( int n )
{



}
```

Looking at factorial() running

- [Whiteboard outline of the execution process when we calculate "factorial(5)"]

Some possible problems

- What happens with a really big number?
- What happens if our test cases are wrong?

Approaching factorial iteratively

- Recursive problems can always be solved using an "iterative" approach (i.e., using loops)
- How would we write factorial() without recursion?
  - What problems does this solve?

Another recursion example

- Fibonacci numbers
  - Basic formula:
    - Fib(0) = 0
    - Fib(1) = 1
    - Fib(n) = Fib(n-1) + Fib(n-2)
  - Example:
    - Fib(4) = Fib(3) + Fib(2)
      = (Fib(2) + Fib(1)) + (Fib(1) + Fib(0))
      = ((Fib(1) + Fib(0)) + Fib(1)) + (Fib(1) + Fib(0))
      = ((1 + 0) + 1) + (1 + 0)
      = (1 + 1) + 1
      = 2 + 1
      = 3

Challenge #1

- Write a recursive implementation of Fibonacci's algorithm.
- Write an iterative implementation.
- Which do you think is more efficient? (Why?)
  - Hint #1: Both implementations will be doing most of the same core *math* for the algorithm.
  - Hint #2: Think about what they do *differently*, and how "expensive" these operations may be....

Challenge #2

- Binary searches can also be written recursively
- Finish the code on the next slide
  - The `binarySearch()` method is to be called by users
  - The `binarySearch2()` method does the work, and is recursive

Challenge #2

```
public int binarySearch( int [] data, int value )
{
    return binarySearch2( data, 0, data.length, value );
}

private int binarySearch2(
            int [] data, int first,
            int numElements, int value )
{
        if ( numElements <= 0 )
            return NOT_FOUND;
        int middleIndex = (numElements / 2) + first;

        //
        // What happens next?
        //









}
```

Tail recursion

- "Tail recursion" means that the last thing that a function does is call itself recursively.
  - Our implementation of factorial was not tail recursive, since it did a multiplication before returning
- Tail recursion usually allows the compiler to perform some optimizations on the code, improving its performance.
  - Tail-recursive functions are also the easiest to convert to iterative implementations

Tail recursive factorial

```
public long factorial( int n )
{
   return factorial( n, 1 );
}

private long factorial( int n, long total )
{
   if ( n < 0 )
        throw new IllegalArgumentException();
   if ( n == 0 || n == 1 )
        return total;
   else {
        total *= n;
        return factorial( n-1, total );
   }
}
```

Tail recursion details

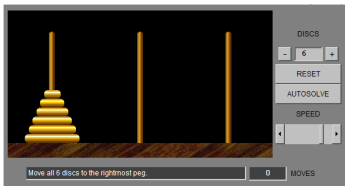
- Tail recursion is often very easy to convert into an iterative (i.e., looping) solution
- It is usually very inefficient (compared to iterative solutions), and should be avoided whenever possible.

When to use recursion

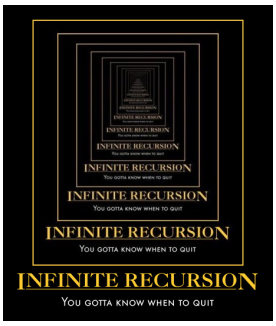
- You want to consider using recursion if:
  - A recursive approach is a natural and obvious solution to the problem, **and**
  - A recursive solution won't result in extra work, **and**
  - An iterative solution will be much more complicated than the recursive solution, **and**
  - The depth of the recursion won't be too great
- Given these criteria, was factorial() a good fit for a recursive approach?
  - Review the criteria, and defend your reasoning

A good example of recursion

- The Towers of Hanoi
  - Example: http://www.ypass.net/java/toh/



Another bad example of recursion



Any Questions?