Today’s topics

- Programming
- Recursion
- Invariants

Reading
- Great Ideas, p. 180-186
- Brooksheer, Section 5.5 6.3

Upcoming
- Copyrights, patents, and digital media
- Security

Solving Problems Recursively

- Recursion is an indispensable tool in a programmer’s toolkit
  - Allows many complex problems to be solved simply
  - Elegance and understanding in code often leads to better programs: easier to modify, extend, verify
  - Sometimes recursion isn’t appropriate, when it’s bad it can be very bad—every tool requires knowledge and experience in how to use it
- The basic idea is to get help solving a problem from coworkers (clones) who work and act like you do
  - Ask clone to solve a simpler but similar problem
  - Use clone’s result to put together your answer
- Need both concepts: call on the clone and use the result

Fundamentals of Recursion

- Base case (aka exit case)
  - Simple case that can be solved with no further computation
  - Does not make a recursive call
- Reduction step (aka Inductive hypothesis)
  - Reduce the problem to another smaller one of the same structure
  - Make a recursive call, with some parameter or other measure that decreases or moves towards the base case
    - Ensure that sequence of calls eventually reaches the base case
    - “Measure” can be tricky, but usually it’s straightforward
- The Leap of Faith!
  - If it works for the reduction step is correct and there is proper handling of the base case, the recursion is correct.
- What row are you in?

Classic examples of recursion

- For some reason, computer science uses these examples:
  - Factorial: we can use a loop or recursion, is this an issue?
  - Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, ...
    - \( F(n) = F(n-1) + F(n-2) \), why isn’t this enough? What’s needed?
    - Classic example of bad recursion, to compute \( F(6) \), the sixth Fibonacci number, we must compute \( F(5) \) and \( F(4) \). What do we do to compute \( F(5) \)? Why is this a problem?
  - Towers of Hanoi
    - N disks on one of three pegs, transfer all disks to another peg, never put a disk on a smaller one, only on larger
    - Every solution takes “forever” when \( N \), number of disks, is large
  - Reversing strings
    - Append first character after the rest is reversed
**Exponentiation**

- Computing $x^n$ means multiplying $n$ numbers (or does it?)
  - What’s the easiest value of $n$ to compute $x^n$?
  - If you want to multiply only once, what can you ask a clone?

```java
double Power(double x, int n)
// post: returns $x^n$
{
    if (n == 0)
    {
        return 1.0;
    }
    return x * Power(x, n-1);
}
```

- What about an iterative version?

**Faster exponentiation**

- How many recursive calls are made to compute $2^{1024}$?
  - How many multiplies on each call? Is this better?

```java
double Power(double x, int n)
// post: returns $x^n$
{
    if (n == 0)
    {
        return 1.0;
    }
    double semi = Power(x, n/2);
    if (n % 2 == 0)
    {
        return semi*semi;
    }
    return x * semi * semi;
}
```

- What about an iterative version of this function?

**Loop Invariants**

- Want to reason about the correctness of a proposed iterative solution
- Loop invariants provide a means to effectively about the correctness of code

```java
while !done do
{
    // what is true at every step
    // Update/iterate
    // maintain invariant
}
```

**Bean Can game**

- Can contains $N$ black beans and $M$ white beans initially
- Emptied according the following repeated process
  - Select two beans from the can
  - If the beans are:
    * same color: put a black bean back in the can
    * different colors: put a white bean back in the can
  - Player who chooses the color of the remaining bean wins the game
- Analyze the link between the initial state and the final state
- Identify a property that is preserved as beans are removed from the can
  - **Invariant** that characterizes the removal process
Recursive example 1

double power(double x, int n) // post: returns x^n
{
    if (n == 0)
    {
        return 1.0;
    }
    return x * power(x, n-1);
}

Recursive example 2

double fasterPower(double x, int n) // post: returns x^n
{
    if (n == 0)
    {
        return 1.0;
    }
    double semi = fasterPower(x, n/2);
    if (n % 2 == 0)
    {
        return semi*semi;
    }
    return x * semi * semi;
}

Recursive example 3

String mystery(int n)
{
    if (n < 2) {
        return "" + n;
    }
    else {
        return mystery(n / 2) + (n % 2);
    }
}