Recursion
(isn’t so hard)

Memory model
Examples
Depth-first search

Recursion: Basis and Progress

A recursive method may call itself on a smaller problem

Always divided into basis and progress:
  Basis: Solve a small problem directly
  Progress: Break off a smaller problem to solve with a recursive call

Factorial

n! = 1 \times 2 \times 3 \times 4 \times \ldots \times n

Recursive definition:

n! = \begin{cases} 
1 & \text{if } n < 2 \\
n \times (n-1)! & \text{if } n \geq 2 
\end{cases}

From definition to method ...

Recursive definition: 

\begin{align*}
  n! &= 1 & \text{if } n < 2 \\
  n! &= n \times (n-1)! & \text{if } n \geq 2 
\end{align*}

Java code:

```java
static int fact(int n) {
    if (n < 2) { return 1; }
    return n \times fact(n - 1);
}
```
fact(5):
    return 5 * fact(5 - 1)
fact(4):
    return 4 * fact(4 - 1)
fact(3):
    return 3 * fact(3 - 1)
fact(2):
    return 2 * fact(2 - 1)
fact(1):
    return 1
Recursion vs Loop

static int fact(int n) {
    if (n < 2) {
        return 1;
    }
    return n * fact(n - 1);
}

Recall Inductive Loop Design

while (not basis case) {
    make the problem a little smaller;
}
solve the basis case;

Almost the same

foo(problem) {
    if (basis case) {
        return the solution;
    } else {
        foo( smaller problem);
    }
}

Facts for computing \(gcd(a,b)\)

gcd(n,0) = n
    because \(n \times 0 = 0\), for all \(n\)
    base case

gcd(a,b) = gcd(b,a)
    (does not make the problem smaller
     ... use with care)

gcd(a, b) = gcd(b,a mod b) if \(a > b\)
    progress case for loop or recursion
Recursive Method

```cpp
int gcd(a, b) {
    if (b == 0) { // base case
        return a;
    }
    // progress case
    return gcd(b, a % b);
}
```

Compare to the loop

```cpp
int gcd(a, b) {
    if (b == 0) {
        return a;
    }
    // gcd(a, b) = (a mod b, a)
    return gcd(b, a % b);
}
```

while (b > 0) {
    // gcd(a, b) = (a mod b, a)
    int temp = b;
    b = a % b;
    a = temp;
}
```

return a;

Week 6: Flood fill

Starting at opening 3, how do we find all the spaces to color yellow?

Flpod fill algorithm (recursion)
Where do variables and objects live?
Let’s lift the lid and peek inside the Java virtual machine ...

(very similar to Python interpreter, and to most language run-time systems)

The Activation Stack
A sequence of “activation records” (a.k.a. stack frames)
One for each method invocation
Holds local variables
The actual contents of “primitive” types
(int, boolean, float, double, ...) References to objects and arrays
(String, Muppet, int [, ... )
The stack grows and shrinks

Stack

```
main
  x: 42
foo
  m: 11
bar
  x: 12
  y: 13
```

```
foo ( ... ) {
  m: ...
  ...
  bar(12);
}
void bar(int x) {
  y: int = 13;
}
```

Recursion creates multiple activation records for a method

Stack

```
main
  x: 42
foo
  m: 11
bar
  x: 12
  y: 13
```

```
foo ( ... ) {
  m: ...
  ...
  bar(12);
}
void bar(int x) {
  y: int = 13;
  if ( ... ) {
    bar (y-3);
  }
}
```

Consequences ...

When your program crashes, you get a “stack trace”:
  Local variables and executing line from each current activation record

Very deep recursion can use up the stack
  If you have an infinite recursive loop, this is how it will crash

Objects live in the heap

Strings are objects
Arrays are similar

The activation record (stack frame) holds a reference (pointer) to the object
A closer look at the object...

Objects contain

A descriptor (reference to the class)
Fields (instance variables)
class Muppet {
    int x;
    static int y;
    foo() { ... }
}

Notes:
You see what m.foo() means ... find the foo() in m's Class record

Instance variables live in the object. Static (class) variables live in the class.
Summary: Where things live

The stack contains an activation record for each active method execution
  Grows and shrinks
  Contains local variables, etc
The heap contains objects (also arrays, classes)
  Things with unknown sizes and lifetimes
Object references are pointers into the heap