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 (and sometimes more efficient!!)
  - 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

Print words entered, but backwards

- Can use a vector, store all the words and print in reverse order
  - The vector is probably the best approach, but recursion works too

```java
void printReversed() { // some I/O details omitted
    String word;
    word = console.readLine();
    if (word.length() > 0) { // get something?
        printReversed(); // print the rest reversed
        System.out.println(word); // then print the word
    }
} // somewhere in main
---.printReversed();
```

- The function `printReversed` reads a word, prints the word only after the clones finish printing in reverse order
- Each clone has its own version of the code, its own `word` variable

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) {
    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 computer $2^{1024}$?
  - 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?
Keys to Recursion

- Recursive functions have two key attributes
  - There is a base case, sometimes called the halting or exit case, which does not make a recursive call
    - See print reversed, exponentiation, factorial for examples
  - All other cases 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

- Example: sequential search in a vector
  - If first element is search key, done and return
  - Otherwise look in the “rest of the vector”
  - How can we recurse on “rest of vector”?

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

Fibonacci: Don’t do this recursively

```java
long recFib(int n)
// precondition: 0 <= n
// postcondition: returns the n-th Fibonacci number
{
    if (0 == n || 1 == n) {
        return 1;
    }
    else {
        return recFib(n-1) + recFib(n-2);
    }
}
```

- How many clones/calls to compute \( F(5) \)?

- How many calls of \( F(1) \)?
- How many total calls?

- consider caching code

Towers of Hanoi

- The origins of the problem may be in the far east
- Move \( n \) disks from one peg to another in a set of three
```java
void move(int from, int to, int aux, int numDisks)
// pre: numDisks on peg # from,
//      move to peg # to
// post: disks moved from peg 'from'
//       to peg 'to' via 'aux'
{
    if (numDisks == 1) {
        System.out.println("move " + from + " to " + to);
    }
    else {
        move(from,aux,to, numDisks - 1);
        move(from,to,aux, 1);
        move(aux,to,from, numDisks - 1);
    }
}
```

- Peg#1    #2          #3
**What’s better: recursion/iteration?**

- There’s no single answer, many factors contribute
  - Ease of developing code assuming other factors ok
  - Efficiency (runtime or space) can matter, but don’t worry about efficiency unless you know you have to
- In some examples, like Fibonacci numbers, recursive solution does extra work, we’d like to avoid the extra work
  - Iterative solution is efficient
  - The recursive inefficiency of “extra work” can be fixed if we remember intermediate solutions: instance variables
- Instance variable: maintains value over all function calls
  - Local variables created each time function called

**Fixing recursive Fibonacci**

```java
long[] storage = new long[31];
long recFib(int n) {
   // pre: 0 <= n <= 30
   // post: returns the n-th Fibonacci number
   Arrays.fill(storage, 0);
   return recF(n);
}
long recF(int n){
   if (0 == n || 1 == n) return 1;
   else if (storage[n] != 0) return storage[n];
   else {
      storage[n] = recF(n-1) + recF(n-2);
      return storage[n];
   }
}
```

- What does storage do? Why initialize to all zeros?
  - Instance variables initialized first time function called
  - Maintain values over calls, not reset or re-initialized

**Thinking recursively**

- Problem: find the largest element in a vector
  - Iteratively: loop, remember largest seen so far
  - Recursive: find largest in [1..n], then compare to 0th element

```java
double max(double[] a)
   // pre: a contains a.length elements, 0 < a.length
   // post: return maximal element of a
   {
      int k;
      double max = a[0];
      for(k=0; k < a.size(); k++) {
         if (max < a[k]) max = a[k];
      }
      return max;
   }
```

- In a recursive version what is base case, what is measure of problem size that decreases (towards base case)?

**Recursive Max**

```java
double recMax(double[] a, int first)
   // pre: a contains a.length elements, 0 < a.length
   // first < a.length
   // post: return maximal element a[first..length-1]
   {
      if (first == a.length-1){ // last element, done
         return a[first];
      }
      double maxAfter = recMax(a, first+1);
      if (maxAfter < a[first]) return a[first];
      else return maxAfter;
   }
```

- What is base case (conceptually)?
- We can use recMax to implement max as follows
  ```java
  return recMax(a, 0);
  ```
Recognizing recursion:

```java
void change(int[] a, int first, int last) {
    if (first < last) {
        int temp = a[first]; // swap a[first], a[last]
        a[first] = a[last];
        a[last] = temp;
        change(a, first+1, last-1);
    }
}
// original call (why?): change(a, 0, a.length-1);
```

- What is base case? (no recursive calls)
- What happens before recursive call made?
- How is recursive call closer to the base case?

More recursion recognition

```java
int value(int[] a, int index) {
     // pre: ??
     // post: a value is returned
     if (index < a.length) {
         return a[index] + value(a,index+1);
     } else {
         return 0;
     }
}
// original call: val = value(a,0);
```

- What is base case, what value is returned?
- How is progress towards base case realized?
- How is recursive value used to return a value?
- What if a is vector of doubles, does anything change?

Grids, vectors, matrices

```java
double[][] grid = new double[20][30];
for(int k=0; k < grid.length; k++) {
    grid[k][5] = 4.0;
}
```

- How is # rows specified? Columns?
  - Possible to do this iteratively, but hard to get right
  - Simple recursive solution

- Twenty rows

Blob Counting: Recursion at Work

- Blob counting is similar to what’s called Flood Fill, the method used to fill in an outline with a color (use the paint-can in many drawing programs to fill in)
  - To write this program we’ll used a class CharBitMap which represents images using characters.
Counting blobs, the first slide

How many blobs are there? Blobs are connected horizontally and vertically, suppose a minimum of 10 cells in a blob

What if blob size changes?

Identifying Larger Blobs

The class Blobs makes a copy of the CharBitMap and then counts blobs in the copy, by erasing noisy data (essentially)

What if blob size changes?

Identifying smaller blobs

What might be a problem if there are more than nine blobs?

What does static mean, values defined in Blobs?

How does the matrix work

Issues that arise in studying code

We’ll study these concepts in more depth, a minimal understanding is needed to work recursive blob find
Recursive helper functions

- Client programs use `findBlobs` to find blobs of a given size in a `CharBitMap` object
  - This is a recursive function, private data is often needed/used in recursive method parameters
  - Use a helper function, not accessible to client code, use recursion to implement member function

- To find a blob, look at every pixel, if a pixel is part of a blob, identify the entire blob by sending out recursive clones/scouts
  - Each clone reports back the number of pixels it counts
  - Each clone “colors” the blob with an identifying mark
  - The mark is used to avoid duplicate (unending) work

Conceptual Details of BlobFill

- Once a blob pixel is found, four recursive clones are “sent out” looking horizontally and vertically, reporting pixel count
  - How are pixel counts processed by clone-sender?
  - What if all the clones ultimately report a blob that’s small?

- In checking horizontal/vertical neighbors what happens if there aren’t four neighbors? Is this a potential problem?
  - Who checks for valid pixel coordinates, or pixel color?
  - Two options: don’t make the call, don’t process the call

- Non-recursive methods takes care of looking for blob(s), then filling/counting/unfilling blobs
  - How is unfill/uncount managed?

Saving blobs

- In current version of `findBlobs` the blobs are counted
  - What changes if we want to store the blobs that are found?
  - How can clients access the found blobs?
  - What is a blob, does it have state? Behavior?
  - What happens when a new minimal blob size is specified?