Review of Recursion with Big-Oh

- 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

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
  - 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

- Most of you have seen the following before in CompSci 6
  - This time, go faster
  - Concentrate on Recurrence Relations and Big-Oh
  - Should now be able to make a more sophisticated evaluation

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

- Recurrence Relations? Big-Oh? Space?

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
    /** @return x^n */
    double power(double x, int n){
      if (n == 0){
        return 1.0;
      }
      return x * power(x, n-1);
    }
    ```
  - Recurrence Relations? Big-Oh?
Faster exponentiation

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

```c
/** @return x^n */
double power(double x, int n){
    if (n == 0) {
        return 1.0;
    }
    double semi = power(x, n/2);
    if (n % 2 == 0) {// is n even?
        return semi*semi;
    }
    return x * semi * semi; { //must be odd
}
```

- Recurrence Relations? Big-Oh?

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

```c
/** @param n >= 0
 * @return n-th Fibonacci number */
long recFib( int n){
    if (0 == n || 1 == n) {
        return 1;
    }else {
        return recFib(n-1) + recFib(n-2);
    }
} How many calls of F(1)?
```

- How many clones/calls to compute $F(5)$?
- Recurrence Relations? Big-Oh? consider caching code

Towers of Hanoi

- Move $n$ disks from one peg to another with restrictive rules

```c
/** disks moved from peg 'from'
 * to peg 'to' using peg 'aux'
 * @param numDisks on peg # from
 * @param from source peg #
 * @parm to target peg #
 * @param aux peg # for parking */
void move(int from, int to, int aux, int numDisks) {
    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);
    }
}
```

- Recurrence Relations? Big-Oh?
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
/** @param n >= 0 and n <= 30
 * @return the n-th Fibonacci number */
long recFib(int n) {
    long[] mem = new long[31];
    Arrays.fill(mem, 0);
    return recF(n, mem);
}
long recF(int n, long[] mem){
    if (0 == n || 1 == n) return 1;
    else if (mem[n] != 0) return mem[n];
    else {
        mem[n] = recF(n-1, mem) + recF(n-2, mem);
        return mem[n];
    }
}
```

- What does `mem` do? Why initialize to all zeros?
  - Recurrence too complicated for our simple methods

Recursive Max

```java
/** @param a contains a.length elements, 0 < a.length 
 * @param first < a.length is index to start 
 * @return maximal element of a[first..length-1]
 */
double recMax(double[] a, int first) {
    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);
  ```
- Recurrence Relations? Big-Oh?

Recognizing recursion:

```java
/** a is changed */
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)
- Recurrence Relations? Big-Oh?