In order to understand recursion, you must first understand recursion. That’s a joke, son.

1. Introduction
   a. Definitions
      • “Recursion” as in recurrence (recur); that which happens again
      • Recursion – defining something in terms of itself
      • Most programming languages allow methods/functions to call *themselves*
   b. Basics
      • Cannot be infinite; need at least one *base case* that terminates all subsequent recursive calls
      • Some computation
      • Recursive call that works toward the base case(s).
   c. Explanation
      • When a method is called, the return point is placed on a *call stack*
      • Local variables must be saved in a stack frame so that when control is returned to the calling method, they are still available
      • The deeper the method calls, the deeper the stack
      • Recursion is no different; it is just the same method that gets called
      • Deep recursion runs the risk of *stack overflow*

2. Examples (see classes in unl.cse.recursion)
   a. Factorial
      • Definition
      • Example
      • Recursive code demo
   b. Fibonacci Sequence
      • Definition: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
      • Recursive definition
      • Code demo
   c. Towers of Hanoi
      • Problem description: move all discs from first peg to third; can use any peg, but cannot place larger discs on top of smaller discs, move one disc at a time
      • Recursive idea: If we were able to move n-1 smaller discs to B, move largest to C, move n-1 smaller discs B -> C
   d. GCD
      • Definition
      • Example
Recursive definition
Code demo (Base case, recursive call)

e. Adaptive Quadrature
- Numerical Integration of a function
- Rectangle Rule/Trapezoidal rule
- Drawback: function can vary greatly; but rules use uniform subdivisions
- Idea: keep subdividing until the difference in function values is “nearly” equal

f. Advanced Examples
- Binomial coefficients
- Quick Sort
- Merge Sort
- Tree Walking

3. Other Issues
a. Advantages & Disadvantages
   - Advantages
     - Cleaner, simpler code
     - Simpler, more readable (more intuitive)
     - Lends itself to a Divide & Conquer strategy
     - Example: Traversing a Linked List or a tree (to come)
   - Disadvantages
     - Method calls result in “context switching” and grows the program stack
     - May lead to stack (or heap) overflow
     - Resource hog (time and memory)
     - Recursion may be performing redundant work (Fibonacci example)
   - Recursion is unnecessary
     - Some languages don’t even support recursion
     - Recursion can usually be replaced with some iterative algorithm
     - Can use a Stack data structure as an alternative
     - Can use memoization to avoid duplicate calls (tableau or map data structures):
       - When each function call is computed, you save it in a tableau (or map)
       - Before making a function call, you check the tableau to see if it has already been computed; use the value if available; call the function if not

b. Tail Recursion
   - Definition: a method is tail-recursive if it reuses its stack frame
   - Nothing for the method to do after the recursive call except return its value
   - Minimizes the stack frame, reduces possibility of a stack overflow
   - Some languages/compilers will unfold tail-recursive code into iterative code
   - Example: Factorial (Code: unl.cse.recursion)
c. Analysis of Recursive algorithms

- Setting up a recurrence relation:
  - Define a function that represents the number of times the elementary operation gets executed with respect to the input n: \( T(n) \)
  - Base cases of the recursive algorithm can be handled directly
  - Otherwise, \( T(n) \) can be expressed in terms of its recursive calls
    - There is some non-recursive cost (number of times the elementary operation is executed in one function call)
    - Plus the contribution of recursive calls (on a smaller input size)
  - This results in a recurrence relation

- Solving recurrence relations: a topic for CSCE 235

- Master Theorem:
  If \( T(n) = aT(n/b) + O(n^k) \) then \( T(n) \) is
  \( O(n^{\log_b(a)}) \) if \( a > b^k \)
  \( O(n^k \log(n)) \) if \( a = b^k \)
  \( O(n^k) \) if \( a < b^k \)