Recursion

- A problem solving technique where an algorithm is defined in terms of itself
- A recursive method is a method that calls itself
- A recursive algorithm breaks down the input or the search space and applies the same logic to a smaller and smaller piece of the problem until the remaining piece is solvable without recursion.
- Sometimes called “divide and conquer”

Recursion vs. Iteration

- in general, any algorithm that is implemented using a loop can be transformed into a recursive algorithm

- moving in the reverse direction is not always possible unless you maintain an additional data structure (stack) yourself.

Recursion Analysis

- in general, recursive algorithms are
  - more efficient
  - more readable (but occasionally quite the opposite!)
  - more “elegant”

- side effects
  - mismanagement of memory
  - “over head” costs

Recursion Components

- Solution to the “base case” problem
  - for what values can we solve without another recursive call?’
- Reducing the input or the search space
  - modify the value so it is closer to the base case
- The recursive call
  - Where do we make the recursive call?
  - What do we pass into that call?

How recursion works

When a method calls itself – it is just as if that method is calling some other method. It is just a coincidence that the method has the same name, args and code. A recursive method call creates an identical copy of the calling method and everything else behaves as usual.

Think of the method as a rectangle containing that method's **code and data, and recursion is just a layering or tiling of those rectangles with information passing to with each call and information returning from each call as the method finishes.

(** code is not actually stored in the call stack)

GCD Algorithm

given two positive integers X and Y,

where X >= Y,

the GCD(X,Y) is

- equal to Y if X mod Y = = 0
  - else
- equal to the GCD(Y, X mod Y)

- Algorithm terminates when the X % Y is zero.
- Notice that each time the function calls it self, the 2nd arg gets closer to zero and must eventually reach zero.

What is the output of this program?

```
public void  foo( int x)
{
   if (x ==0)
            return;
   else
    {       System.out.println( x );
            foo( x - 1 );
    }
}
public static void main( String args[])
{
   foo( 7 );
}
```

** Identify the Base case, recursive call and reduction / modification of the input toward the base case.

What is the output of this program?

```
public int foo( int x)
{
   if (x ==0)
            return 0;
   else
            return x + foo(x-1);
}
public static void main( String args[])
{
            System.out.println( foo(7) );
}
```

** Identify the Base case, recursive call and reduction / modification of the input toward the base case.

What is the output of this program?

```
public int foo( int x, int y)
{
   if (x == 0)
            return y;
   else
            return foo( x-1, y+1 );
}
public static void main( String args[] )
{
            System.out.println( foo( 3, 4 ) );
}
```

** Identify the Base case, recursive call and reduction or modification of the input toward the base case.

What is the output of this program?

```
public int  foo( int x, int y )
{
   if (x == 0)
            return y;
   else
            return foo( x-1, y+x );
}
public static void main( String args[])
{
            System.out.println( foo( 3, 4 ) );
}
```

Now.. You help me write this

- Write a recursive function that accepts an int and prints that integer out in reverse on 1 line

- What is the base case ?
- How do I reduce the input toward base case ?
- What do I pass to the recursive call ?

One more try!

- Write a recursive function that accepts a string and prints that string out in reverse on 1 line.

- What is the base case ?
- How do I reduce the input toward base case ?
- What do I pass to the recursive call ?

Other Examples ...

- Bad examples (but for illustration/treaching)
  - factorial
  - exponential
  - Fibonacci numbers
  - power

Other Examples ...

- Good examples
  - Towers of Hanoi
  - GCD
  - Eight Queens
  - Binary Search Trees
  - Maze traversal
  - Backtracking (i.e recovery from dead ends)

Tail Recursion optimization

- Recursion can use up a lot of memory very quickly!
- The compiler can generate assembly code that is iterative but guaranteed to compute the exact same operation as the recursive source code.
- It only works if the very last statement in your method is the recursive call. This is tail recursion.
- Java does not tail optimize recursive code.