# Ch 12 and 13 Recursion

## Recursion

The definition of an operation in terms of itself. Solving a problem using recursion depends on solving smaller occurrences of the same problem.

## Recursive Programming

Writing methods that call themselves to solve problems recursively.

- An equally powerful substitute for iteration (loops)
- Particularly well-suited to solving certain types of problems

Every recursive method has two distinct parts:

1. A base case or termination condition that causes the method to end.
2. A nonbase case whose actions move the algorithm toward the base case and termination.

Here is the framework for a simple recursive method:

```
public void recursiveMethod(...)
{
    if (base case)
    {
        <Perform some action>
    }else {
        <Perform some other action>
        recursiveMethod(...);
    }
}
```

The base case typically occurs for the simplest case of the problem, such as when an integer has a value of 0 or 1. Other examples of base cases are when some key is found, or an end-of-file is reached. A recursive algorithm can have more than one base case.

In the else or nonbase case of the framework shown, the code fragment <Perform some other action> and the method call recursiveMethod(...) can sometimes be interchanged without altering the net effect of the algorithm. Be careful though. This can sometimes be disastrous.

Example:

```
public void drawLines(int n)
{
    if (n==0)
      System.out.println("That's all, folks!");
    else
    {
      for (int i=0; i<=n; i++)
        System.out.print("* ");
      System.out.println();
      drawLine(n-1);
    }
}
```

What is the output? drawLines(4);

Example 2:

Write a recursive method pow that accepts an integer base and exponent and returns the base raised to that exponent.
Example: pow(3, 4) returns 81

# Recursion (Continued)



Since every recursive algorithm can be written iteratively, when should one use recursion? Bear in mind that recursive algorithms can incur extra run time and memory. Their major plus is elegance and simplicity of code.

## General Rules for Recursion

1) Avoid recursion for algorithms that involve large local arrays — too many recursive calls can cause memory overflow.

2) Use recursion when it significantly simplifies code.

3) Recursion is especially useful for:

- Branching processes like traversing trees or directories
- Divide-and-conquer algorithms like mergesort and binary search.

A certain type of problem crops up occasionally on the AP exam: using recursion to traverse a two-dimensional array. The problem comes in several different guises, for example,

- A game board from which you must remove pieces.
- A maze with walls and paths from which you must try to escape
- White containers enclosed by black walls into which you must pour paint.

In each case, you will be given a starting position (row, col) and instructions on what to do. The recursive solution typically involves these steps.

Check that the starting position is not out of range
if (starting position satisfies some requirement)
Perform some action to solve problem
RecursiveCall (row+1, col)
RecursiveCall (row-1, col)
RecursiveCall (row, col+1)
RecursiveCall (row, col-1)