Recursion

A *recursive* definition refers to itself.

Factorial:

$n! = n \times (n-1)! \quad \text{for all } n \geq 1$
$0! = 1$

Fibonacci:

$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \quad \text{for all } n \geq 3$
$\text{fib}(1) = 1$
$\text{fib}(2) = 2$

Recursive definitions have two parts:

1. Recursive rule, defining larger instances in terms of smaller ones.
2. Definition of enough small instances to get the rule started.

Leading to recursive design technique:

1. Define n^{th}, pretending that all smaller ones are already defined.
2. Write as many small instances as are needed to make the rule work.

Programming recursively

Same logic applies: recursive rule and solve one or more small instances.

Process:

1) Write the programming interface so that it is clear what the problem is and what size the problem is.

```
int pow(int x, int y)  // return x^y for y >=0. y is problem size
int findLargest(int list[], int n) // return largest in list[0..n-1]
```

Process: Step 2

2) Imagine that somebody wrote a library function for you called `smallerSolver`, which can solve any problem of the same type, but whose size input must be smaller than your own.

```
int smallerPow(int x, int smallerY)  // Only works when new y < old y
int smallerFindLargest(int list[], int smallerN)     // new n < old n
```

Process: Step 3

3) Use the following template for the body of your solution

```
begin
  if (problem size is very small)
    solve the problem directly
  else
    solve the problem using smaller_solver
end
```

[Do the examples on the board]

Process: Step 4

4) Replace all occurrences of `smallerSolver` with the name of the function you just wrote.

Example: priceIsRight (PI 7.12)

Write a method priceIsRight that takes an array of integer guesses and an integer price and returns the largest of the bids that is not larger than the price. If no guess is less or equal to the price return -1.

```java
public static int priceIsRight(int[] bids, int price) {  // PI Ex 7.12
    if (bids.length==1) {
        if (bids[0] <= price) {
            return bids[0];
        } else {
            return -1;
        }
    } else {
        int[] smallerArray = Arrays.copyOfRange(bids,1,bids.length);
        int smallerAnswer = priceIsRight(smallerArray, price);
        if (bids[0] <= price) {
            return Math.max(bids[0], smallerAnswer);
        } else {
            return smallerAnswer;
        }

    }
}
```

A more efficient version

```
public static int priceIsRight(int[] bids, int n, int price) { // Consider only indices 0..n-1
    if (n==1) {
        if (bids[0] <= price) {
            return bids[0];
        } else {
            return -1;
        }
    } else {
        int smallerAnswer = priceIsRight(bids, n-1, price);
        if (bids[n-1] <= price) {
            return Math.max(bids[n-1], smallerAnswer);
        } else {
            return smallerAnswer;
        }
    }
}

public static int priceIsRight(int[] bids, int price) {
    return priceIsRight(bids, bids.length, price);
}
```