Recursion

Definition:

- A function that calls itself is said to be recursive.
- A function **f1** is also recursive if it calls a function **f2**, which under some circumstances calls **f1**, creating a cycle in the sequence of calls.
- The ability to invoke itself enables a recursive function to be repeated with different parameter values.
- You can use recursion as an alternative to iteration (looping).

The Nature of Recursion:

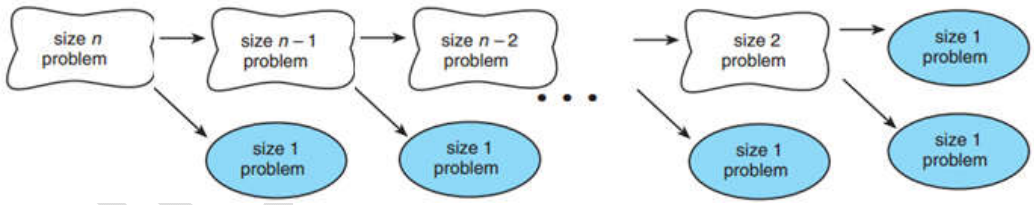
Problems that lend themselves to a recursive solution have the following characteristics:

- One or more simple cases of the problem have a straightforward, non-recursive solution.
- The other cases can be redefined in terms of problems that are closer to the simple cases.
- By applying this redefinition process every time the recursive function is called, eventually the problem is reduced entirely to the simple case(s), which are relatively easy to solve.

The recursive algorithms will generally consist of an "**if** statement" with the following form:

```
if this is a simple case
        solve it
else
        redefine the problem using recursion
```

Illustration:



Example:

Solve the problem of multiplying **6** by **3**, assuming **we only know addition**:

- **Simple case**: any number multiplied by 1 gives us the original number.
- The problem can be split into the two problems:

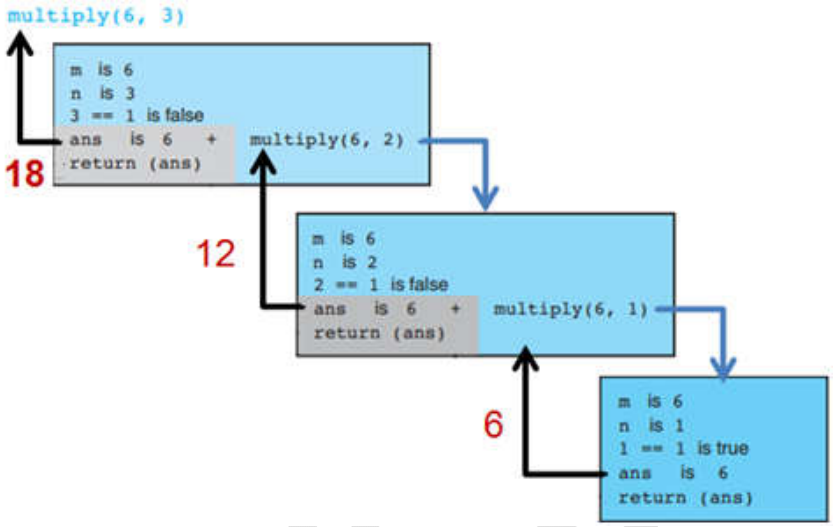
1. Multiply 6 by 2.
1.1 Multiply 6 by 1.
1.2 Add (Multiply 6 by 1) to the result of problem 1.1.
2. Add (Multiply 6 by 1) to the result of problem 1.

Implement this recursively

Tracing a Recursive Function:

- Tracing an algorithm's execution provides us with valuable insight into how that algorithm works.
- By drawing an **activation frame** corresponding to each call of the function.
- An activation frame shows the parameter values for each call and summarizes the execution of the call.

multiply(6, 3):

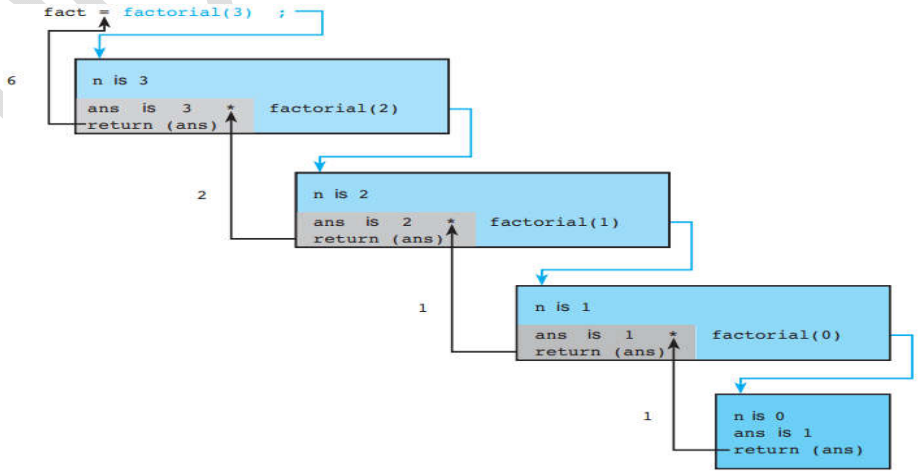


Recursive Mathematical Functions:

- Many mathematical functions can be defined recursively.
- An example is the factorial of n (**n!**):
  - **0!** is 1
  - **n!** is **n * (n– 1)!** , for n> 0
- Thus **4!** is **4 *3!**, which means 4 *3 *2 *1, or 24.

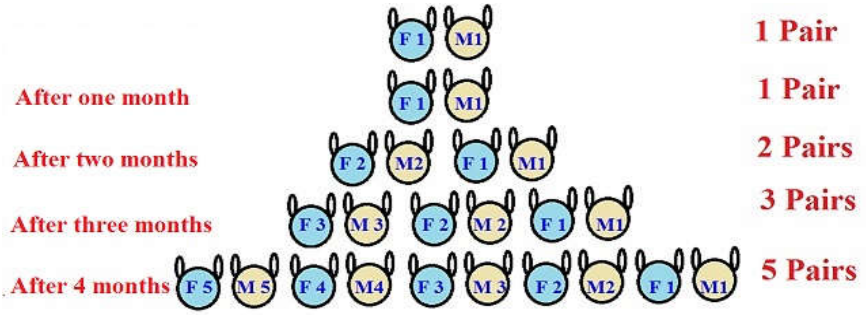
Implement this iteratively and recursively

Tracing the recursive function



Fibonacci Numbers:

Leonardo **Bonacci** (1170 –1250)



- **Problem:**
  - How many pairs of rabbits are alive in month **n**?
  - Recurrence relation:

$$\textbf{rabbit(n)} = \textbf{rabbit}(_{n-1}) + \textbf{rabbit}(_{n-2})$$

```
int n=10, t1=0, t2=1, sum;
for (int i=1; i<=n; ++i){
  System.out.print(t1 +" + ");
  sum = t1 + t2;
  t1 = t2;
  t2 = sum;
}
```

❖ The Fibonacci sequence is defined as:

- Fibonacci **0** is 1
- Fibonacci **1** is 1
- Fibonacci **n** is (**Fibonacci** $_{n-2}$) + (**Fibonacci** $_{n-1}$), for n>1

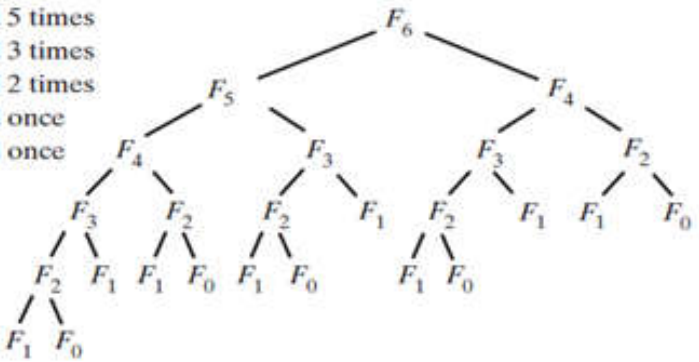
Implement this recursively

Poor Solution to a Simple Problem:

```
Algorithm Fibonacci(n)
if (n <= 1)
   return 1
else
   return Fibonacci(n - 1) + Fibonacci(n - 2)
```

Why is this inefficient? Try F_6

F_2 is computed 5 times
F_3 is computed 3 times
F_4 is computed 2 times
F_5 is computed once
F_6 is computed once



Suggest an efficient recursive solution

Self-Check:

- Write and test a recursive function that returns the value of the following recursive definition:
  - **f(x) = 0** **if x = 0**
  - **f(x) = f(x - 1) + 2** **otherwise**

What set of numbers is generated by this definition?

Recursion Design Guidelines:

- Method must be given an **input value**.
- Method definition must contain **logic** that involves this input, leads to different cases.
- One or more cases should provide solution that does not require recursion.
  - else **infinite loop**
- One or more cases must include a recursive invocation.

Stack of Activation Records:

- Each call to a method generates an activation record.
- Recursive method **uses more memory** than an iterative method.
  - Each recursive call generates an activation record.
- If recursive call generates too many activation records, could cause **stack overflow**.

Recursively Processing an Array:

Starting with array[first]:

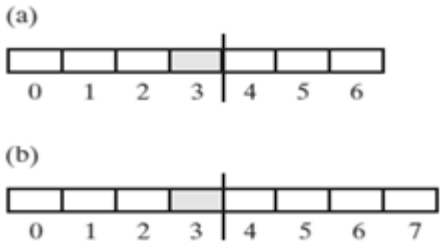
```
public static void displayArray(int array[], int first, int last)
{
   System.out.print(array[first] + " ");
   if (first < last)
      displayArray(array, first + 1, last);
} // end displayArray
```

Starting with array[last]:

```
public static void displayArray(int array[], int first, int last)
{
   if (first <= last)
   {
      displayArray(array, first, last - 1);
      System.out.print (array[last] + " ");
   } // end if
} // end displayArray
```

Processing array from middle:

```
int mid = (first + last) / 2;
```




```
public static void displayArray(int array[], int first, int last)
{
   if (first == last)
      System.out.print(array[first] + " ");
   else
   {
      int mid = (first + last) / 2;
      displayArray(array, first, mid);
      displayArray(array, mid + 1, last);
   } // end if
} // end displayArray
```

Consider
first + (last – first) / 2
Why?