Lecture 7: Recursive Function

Check the Notes pages (section)

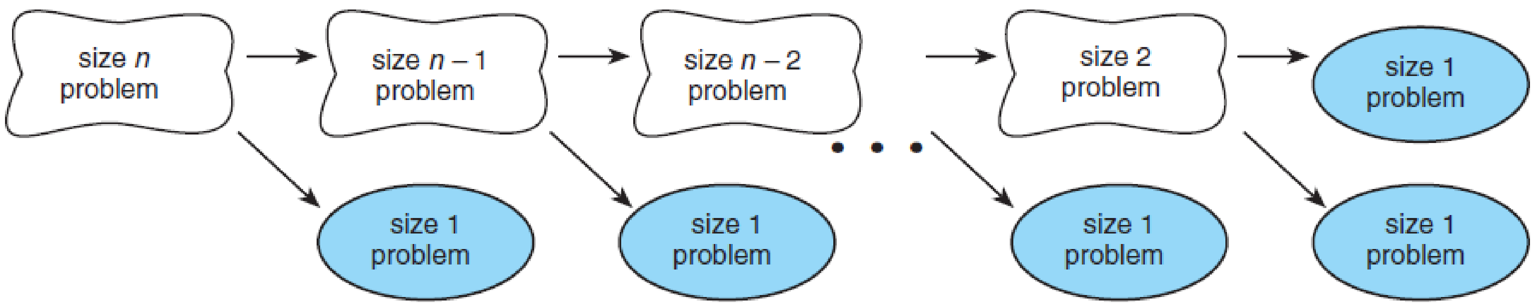
What is Recursion?

- **Recursion** is a **problem-solving approach**, **that can splits a problem into one or more simpler versions of itself**
  - A recursion is when one function calls ITSELF directly or indirectly

- **General Structure:**
  - ***one (or more) base cases***, for which **the result of the function can be determined directly**
  - ***one (or more) recursive cases***, for which **the computation of the result is reduced to the computation of the same function on a smaller/simpler values of the input arguments**
    - ***Recursive cases* must converge towards the *base case(s)***

…Continued

FIGURE 9.1 Splitting a Problem into Smaller Problems



Let's assume that for a particular problem of size *n,* we can split the problem into a problem of size 1, which we can solve (a base/simple case), and a problem of size *n* - 1.

We can split the problem of size *n* - 1 into another problem of size 1 and a problem of size *n* - 2, which we can split further.

If we split the problem *n* - 1 times, we will end up with *n* problems of size 1, all of which we can solve.

Example of a Recursive Function: Factorial

❖ The **factorial :** *fact(n)* for any positive integer *n*, written *n!*, is defined to be the product of all integers between 1 and *n* inclusive:

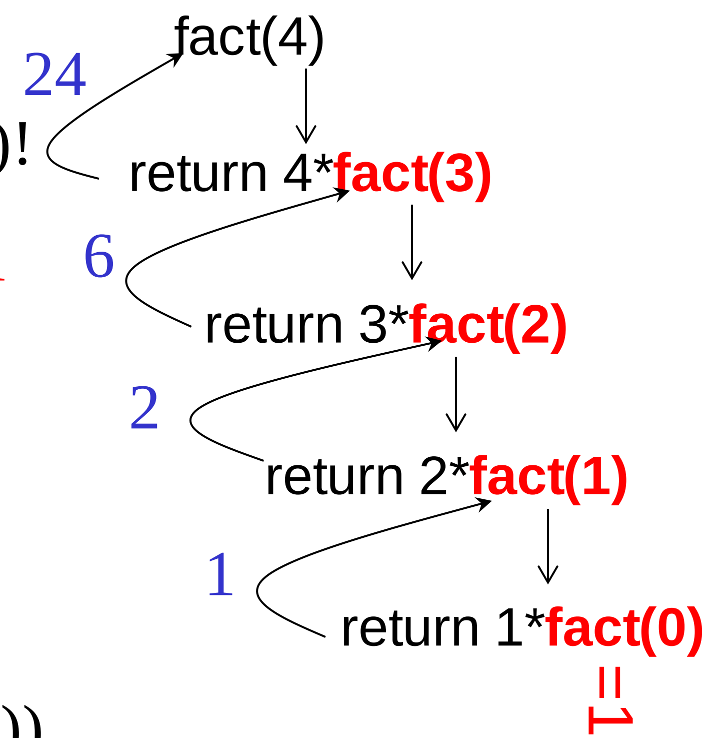
$$fact(n) = n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

❖ We note that: 0! = 1, 1!=1 and n! = n*(n-1)!

hence, fact(n) = n*fact(n-1) & fact(0)=1

❖Let's *calculate* fact(4) ***recursively***:

fact(**4**) = 4 * **fact**(**3**)
= 4 * (3 * **fact**(**2**))
= 4 * (3 * (2 * **fact**(**1**)))
= 4 * (3 * (2 * 1***fact**(**0**))))
= 4* (3 * (2 * 1* 1)))=24



- ❑ As always, **when the function completes, control returns to the function that invoked it** (which is invocation of the same function)

Recursive Function Coding

❖ A **recursive function** has the following general form:

Function(Arguments **)** {

IF *the **base case**,*

*return the simple value // base case or **stopping condition***

ELSE

*call **Function** with simpler version of argument (**recursive expression**)*

}

Factorial Function-Iterative Implementation

$$fact(n) = n! = n\times(n-1)\times(n-2)\times\ldots\times 2\times 1$$

❖ Use a loop to calculate factorial *iteratively*

```
// iteratively
int fact(int n) {
int product = 1;
for (int i = 2; i <= n; i++) {
product = product* i;
}
return product;
}
```

1. 0! = 1; 1!=1

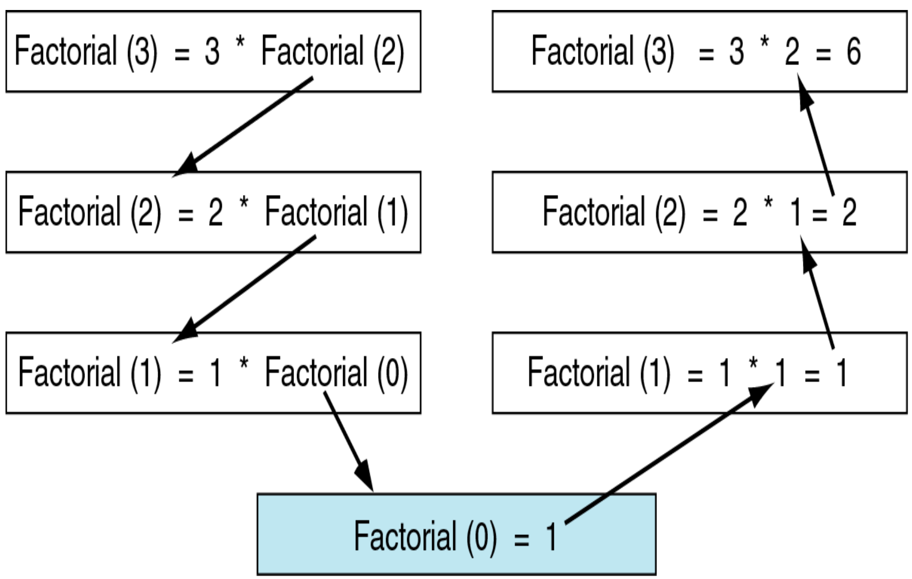
2. Loop

1*2	= 2	:	*first iteration*
2*3	= 6	:	*second iteration*
6*4	= 24	:	*third iteration*
24*5	= 120	:	*forth iteration*

Factorial Function - Recursive Implementation

```
int factorial(int num)
{
if (num == 0)          Base case
   return 0;
else
   return ( num * factorial(num-1));   Recursive case
}
```

fact(0)=1
fact(n) = n*fact(n-1)

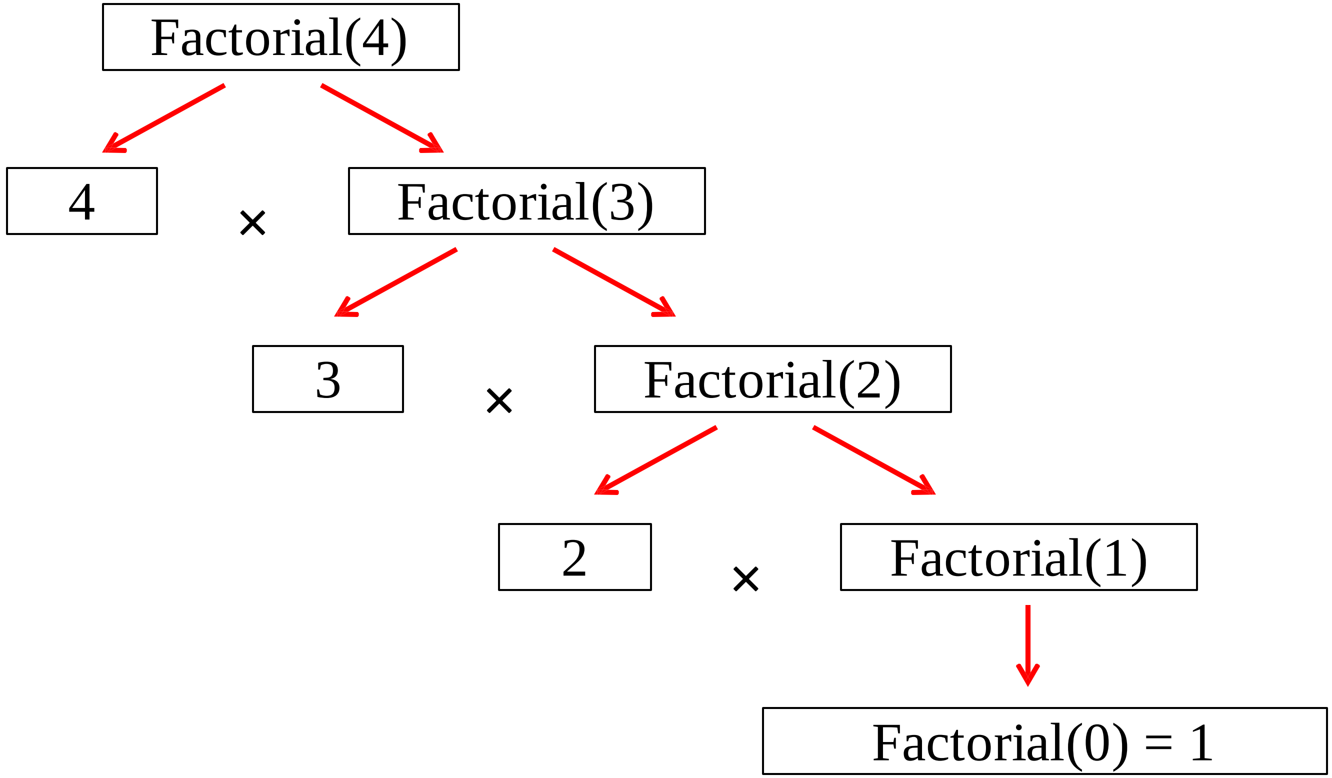


UNIVERSITY OF WOLLONGONG IN DUBAI

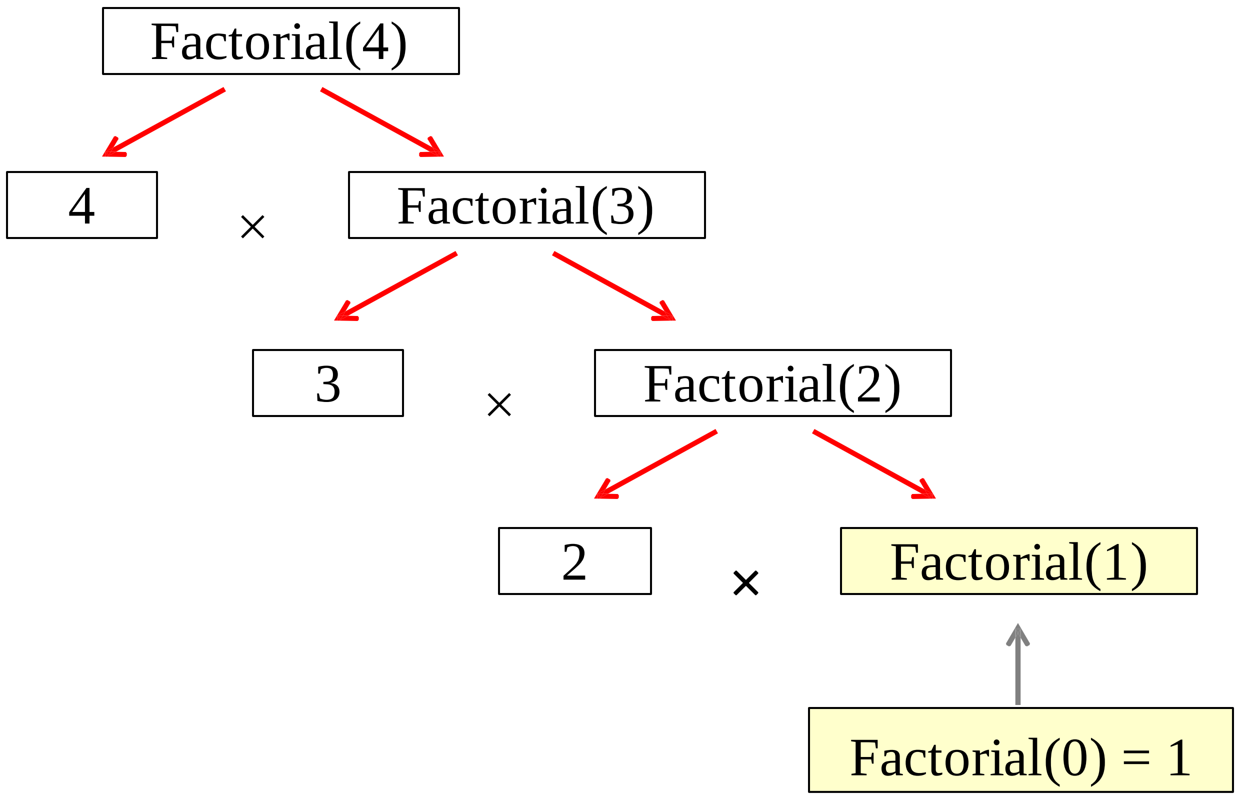
…Continued

fact(0)=1

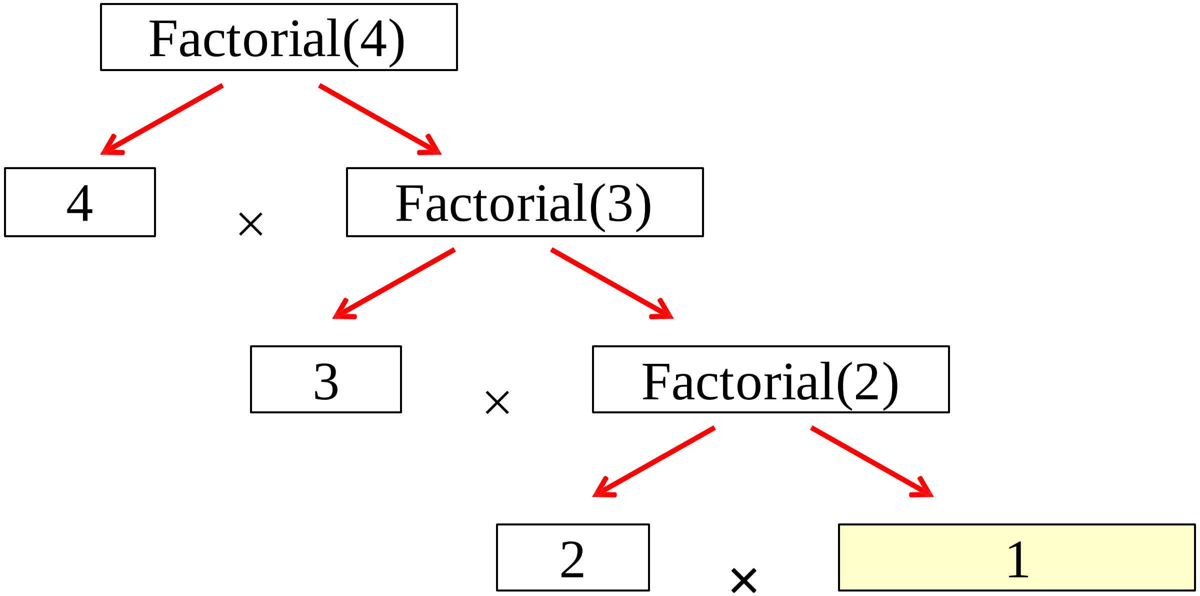
fact(n) = n*fact(n-1)



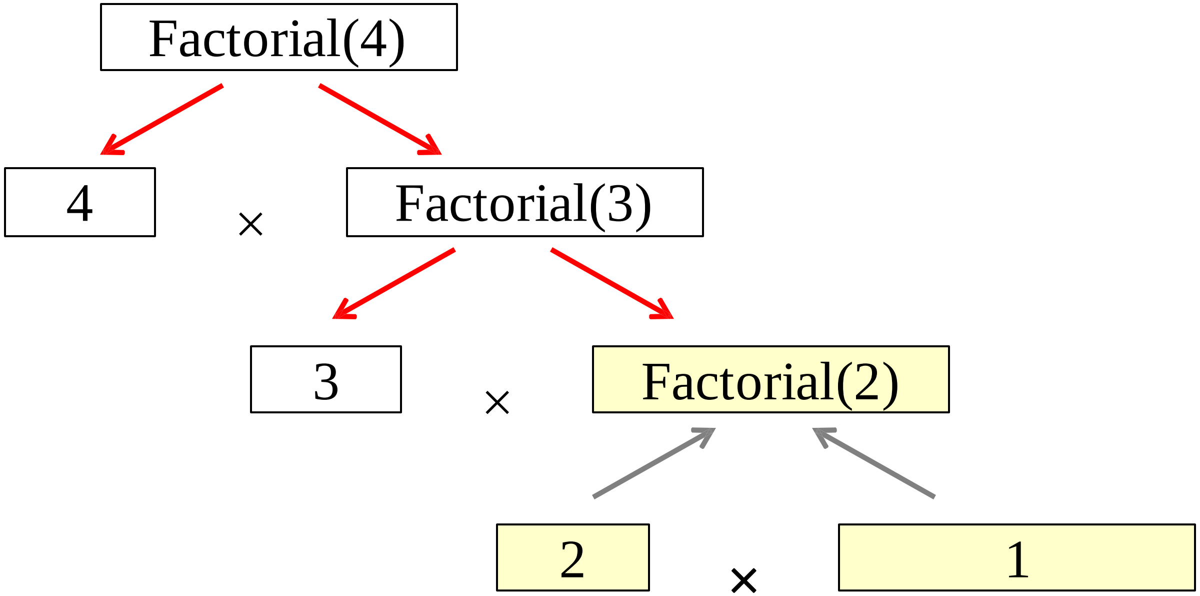
…Continued


Factorial(4)
4
×
Factorial(3)
3
×
Factorial(2)
2
×
Factorial(1)
Factorial(0) = 1

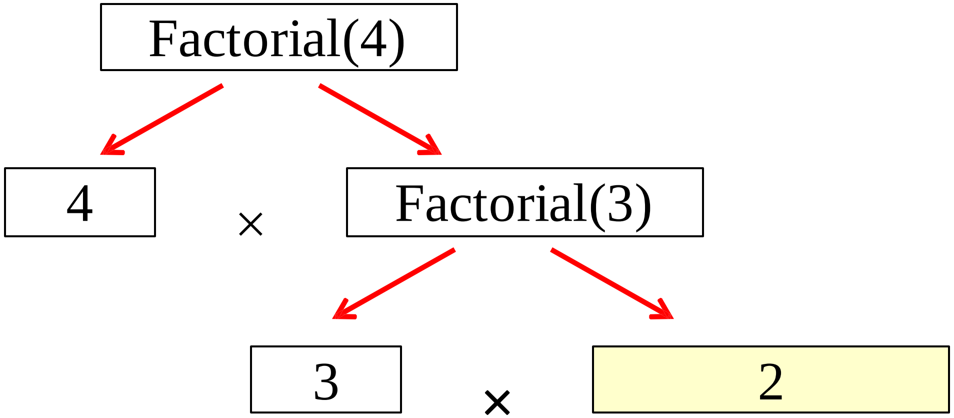
...Continued



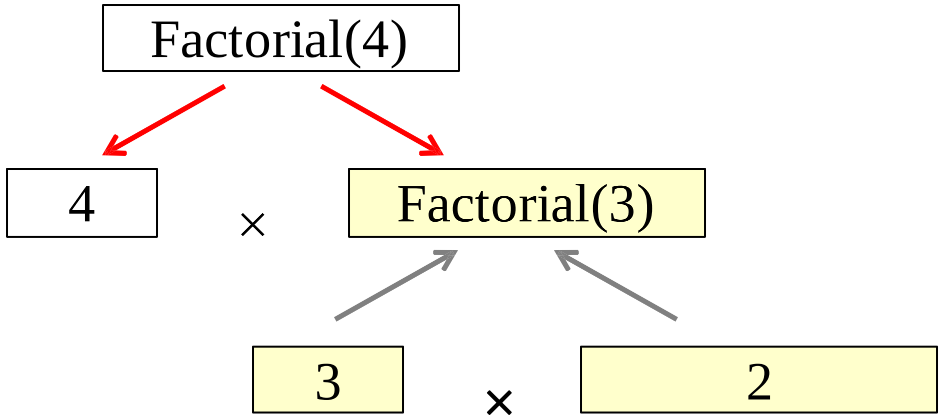
…Continued



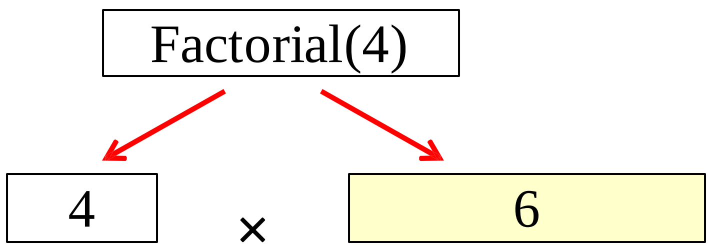
…Continued



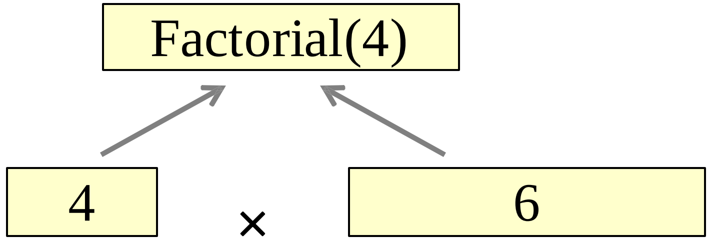
…Continued



…Continued



…Continued



…Continued

24

UNIVERSITY OF WOLLONGONG IN DUBAI

Recursion vs. Iteration

- Most **recursive solutions have corresponding iterative** solutions
  - For example, fact(n) = n! can be calculated with a loop

```
// recursive
int fact(int n) {
if (n == 0) {
return 1;
} else {
return n * fact(n - 1);
}}
```

Condition tests for the <u>*base case*</u> *of your function*

```
// iteratively
int fact(int n) {
int product = 1;
for (int i = 2; i <= n; i++) {
product = product* i;}
return product;
}
```

Condition test for the <u>Loop condition</u> ***to determine whether to exit***

$5! = 5 \times 4 \times 3 \times 2 \times 1$

- **Recursive** solutions **are often *slower*** than **iterative** because of **multiple function invocations.** However, for some problems recursive solutions are often more simple and elegant than iterative solutions

Recursive Function Design

1. Determine the **base case(s).**
2. Then, determine the **general (recursive) case.**
3. Combine the **base case** and **general case** into a function.
4. In combining the **base** and **general cases** into a function, you must pay careful attention to the logic.
   - Each call must reduce the size of the problem and move it toward the **base case**.
   - The **base case**, when reached, **must terminate WITHOUT a function call**. **It must set the** return value.

Examples

- **Example1**: create a function power() to compute x^n, where n is a positive number
  1. Create an iterative version of the function
  2. Create a recursive version of the function
  3. Compare the two solutions

Iterative Solution

$$x^n = \underbrace{x * x * x \ldots * x}_{\text{n times}}$$

```
/* Iterative solution */
double power(double x, int n)
{
    int i;
    double result = x;

    if (n == 0) return 1.0;

    for ( i=1; i < n; i++ )
        result *= x;

    return result;
}
```

Recursive Solution

1. Base case

   $x^0 = 1.0$

`power(x, 0) = 1.0, if (n == 0)`

2. General case

   $x^n = x * x^{n-1}$

   `power(x, n) = x * power(x, n-1)`

```
/* Recursive solution */
double power(double x, int n)
{
   if (n == 0)          /* base case */
       return 1.0;

   else                 /* general case */
       return ( x * power(x, n-1) );
}
```

…Continue

❖ **Example 2**: a recursive function multiply that returns the product of its two arguments m × n; both m and n are strictly positive

Answer: $m \times n = m \times (n-1) + m.$

Base case

$m \times 1 = m$

If we define f(n)=m*n, then f(n)= f(n-1)+m

```
/*
 * Performs integer multiplication using + operator.
 * Pre:   m and n are defined and n > 0
 * Post:  returns m * n
 */
int
multiply(int m, int n)
{
      int ans;

      if (n == 1)
            ans = m;     /* simple case */
      else
            ans = m + multiply(m, n - 1); /* recursive step */

      return (ans);
}
```

Fibonacci

Sequence is 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Fibonacci(1) is 1
Fibonacci(2) is 1
Fibonacci(n) is Fibonacci(n-2) + Fibonacci(n-1) for n > 2

```
/*
 * Computes the nth Fibonacci number
 * Pre: n > 0
 */
int
fibonacci(int n)
{
      int ans;

      if (n == 1 || n == 2)
            ans = 1;
      else
            ans = fibonacci(n - 2) + fibonacci(n - 1);

      return (ans);
}
```

Greatest Common Divisor (gcd)

gcd(*m* , *n*) is *n* if *n* divides *m* evenly

gcd(*m* , *n*) is gcd(*n* , remainder of *m* divided by *n*) otherwise

```
/*
 * Finds the greatest common divisor of m and n
 * Pre: m and n are both > 0
 */
int
gcd(int m, int n)
{
      int ans;

      if (m % n == 0)
            ans = n;
      else
            ans = gcd(n, m % n);

      return (ans);
}
```