BIRZEIT UNIVERSITY

Recursion

By: Mamoun Nawahdah (Ph.D.)
2014/2015

Recursion

- 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 simple cases, which are relatively easy to solve.

The Nature of Recursion

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

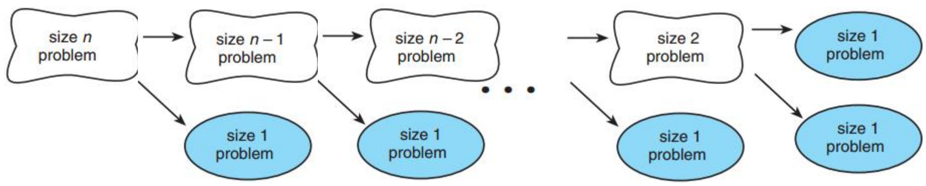
if this is a **simple case**

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 **6** to the result of problem 1.1. ✓
  2. Add **6** to the result of problem 1. ✓

FIGURE 9.2 Recursive Function multiply

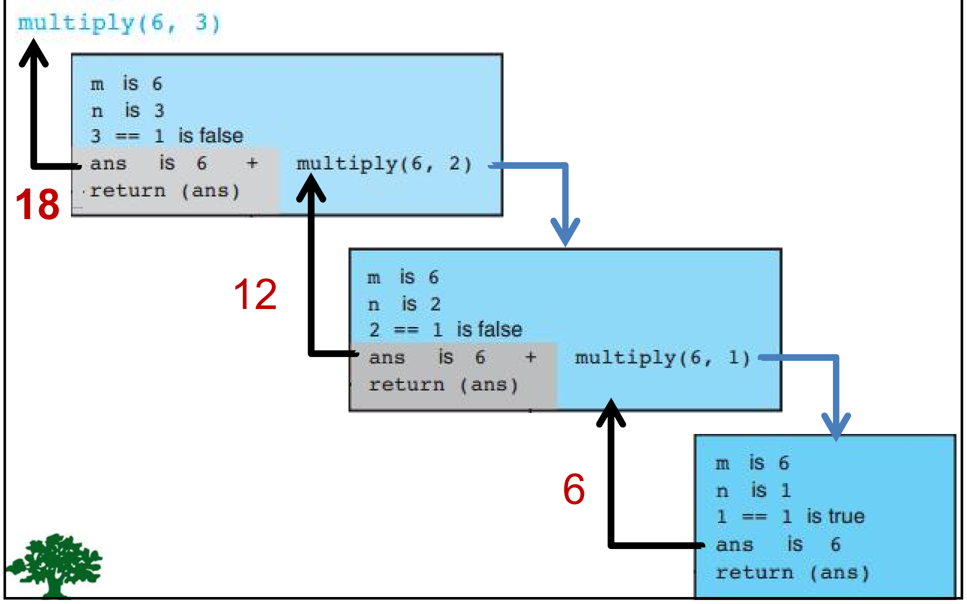
```
/*
 * Performs integer multiplication using + operator.
 * Pre:   m and n are defined and n > 0
 * Post:  returns m * n
 */
int
multiply(int m, int n)
{
    int ans;

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

    return (ans);
}
```

The simplest case is reached when n == 1

Tracing a Recursive Function

- Hand 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** — The simplest case
- **n! is n * (n– 1)! , for n> 0**

❖ Thus 4! is 4 *3!, which means 4 *3 *2 *1, or 24.

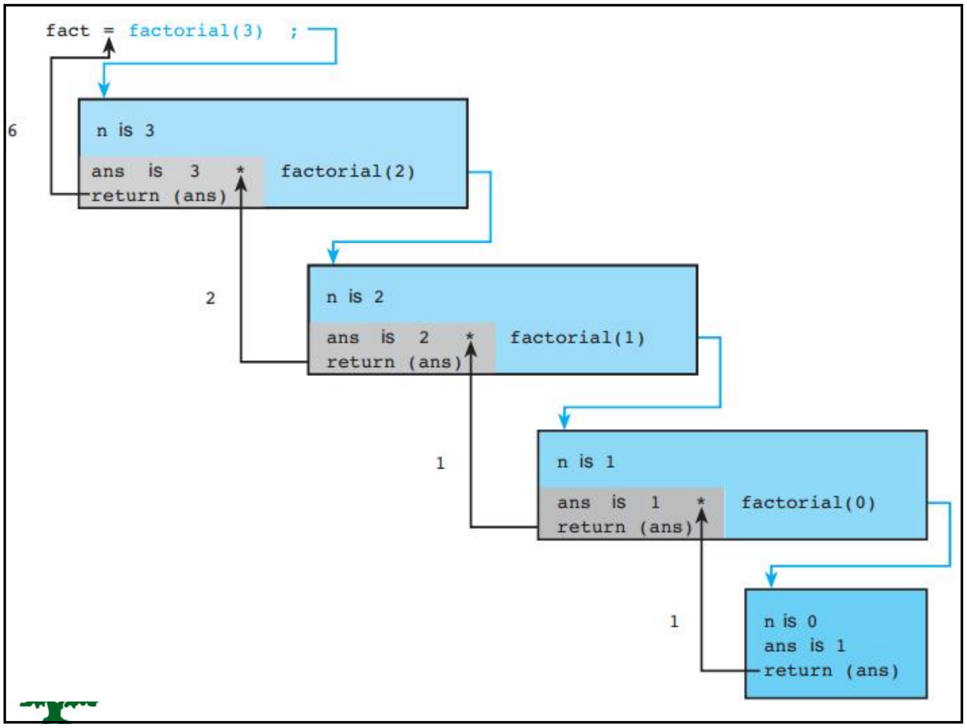
FIGURE 5.7 Function to Compute Factorial

```
/*
 * Computes n!
 * Pre: n is greater than or equal to zero
 */
int
factorial(int n)
{
    int i,         /* local variables */
        product;   /* accumulator for product computation */

    product = 1;
    /* Computes the product n x (n-1) x (n-2) x . . . x 2 x 1 */
    for (i = n; i > 1; --i) {
         product = product * i;
    }

    /* Returns function result */
    return (product);
}
```

FIGURE 9.10 Recursive factorial Function

```
/*
 * Compute n! using a recursive definition
 * Pre: n >= 0
 */
int
factorial(int n)
{
      int ans;

      if (n == 0)
            ans = 1;
      else
            ans = n * factorial(n - 1);

      return (ans);
}
```

The simplest case



Fibonacci Numbers

- The Fibonacci sequence is defined as:
  - Fibonacci 1 is 1
  - Fibonacci 2 is 1
  - Fibonacci n is Fibonacci n−2 + Fibonacci n−1, for n> 2

The simplest cases

Leonardo Bonacci (c. 1170 – c. 1250)

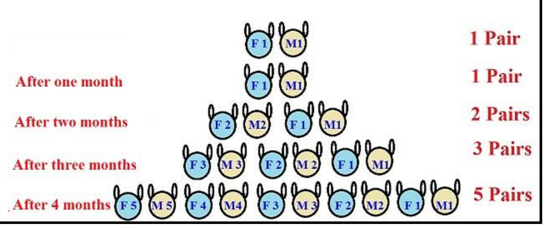


FIGURE 9.13 Recursive Function fibonacci

```c
/*
 * 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);
}
```

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?

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 recursion**
- 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.**

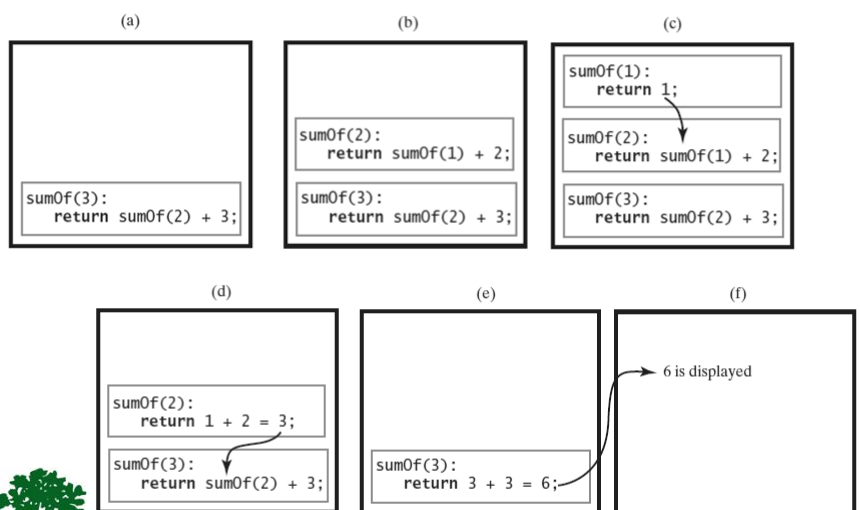
Recursive Methods That Return a Value

Recursive method to calculate $\sum_{i=1}^{n} i$

```
/** @param n  An integer > 0.
    @return  The sum 1 + 2 + ... + n. */
public static int sumOf(int n)
{
   int sum;
   if (n == 1)
      sum = 1;                // Base case
   else
      sum = sumOf(n - 1) + n; // Recursive call

   return sum;
} // end sumOf
```

Tracing a Recursive Method

Tracing the execution of **sumOf(3)**



Recursively Processing an Array

Starting with `array[first]`

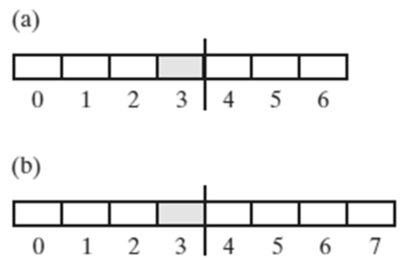
```
public static void displayArray(int array[], int first, int last)
{
   System.out.print(array[first] + " ");
   if (first < last)
      displayArray(array, first + 1, last);
} // end displayArray
```

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
```

Recursively Processing an Array

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



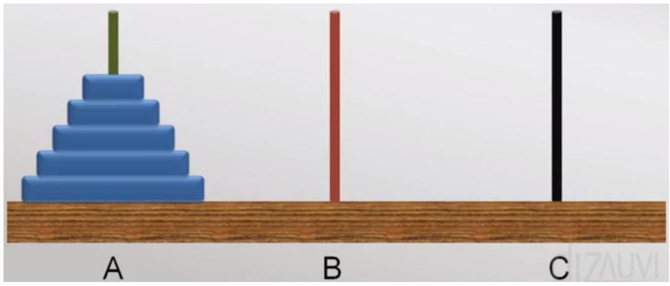
Two arrays with their middle elements within their left halves

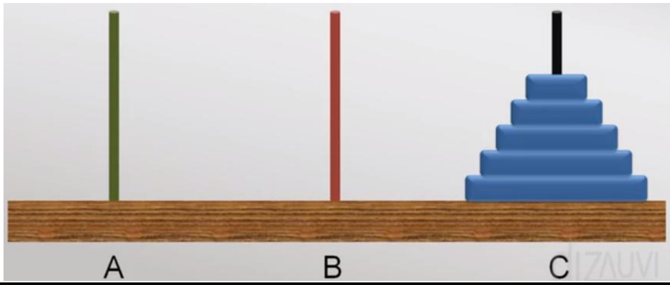
Recursively Processing an Array

```
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
```

Processing array from middle.

Tower of Hanoi





Tower of Hanoi

From

To

Using

# of discs

move(n, A, C, B)

move(n-1, A, B, C)

moving n from A → C

move(n-1, B, C, A)