Introduction to Computers and Programming

Prof. I. K. Lundqvist

Reading: B pp. 181-189, FK pp. 591-621, handout

Lecture 12
Oct 1 2003

PRS 1 -- Scope

1. Nothing is displayed on the screen
2. 0 (zero) is displayed on the screen
3. 7 is displayed on the screen
4. Don’t know

Recursion

- Writing procedures and functions which call themselves
- Involves
  - Solving large problems
  - By breaking them into smaller problems
  - Of identical form
- Eventually, a “trivial” problem (the *base* case) is reached that can be solved immediately

General algorithm

```
• if stopping condition then
     solve simple problem
  else
     use recursion to solve smaller problem(s)
     combine solutions from smaller problem(s)
```

- Iteration
  - Cognitive simple
- Recursion
  - Is not as intuitive
  - Demanding on machine time and memory
  - Sometimes simpler than iteration

Guess a number

- **Problem**: think of a number in the range 1 to N
- **Reworded**:
  - Given a set of N possible numbers to choose from
  - Guess a number from the set
  - If wrong, **guess again**
  - Continue until the number is guessed successfully
- **Recursion** comes into the "**guess again**" stage
  - A set of N-1 numbers remains from which to guess
  - This is a smaller version of the same problem

Factorial

- Write a function that, given n, computes n!

  $n! = 1 * 2 * \ldots * (n-1) * n$
- Example:

  $5! = 1 * 2 * 3 * 4 * 5 = 120$
- Specification:

  Receive: *n*, an integer
  Precondition: *n* >= 0 (0! = 1 and 1! = 1)
  Return: *n!*

Preliminary Analysis

```
function Factorial_Iterative (N : Natural)
                                  return Positive is

   Result : Positive;

begin
   Result := 1;
   for Count in 2 .. N loop
      Result := Result * Count;
   end loop;

   return Result;

end Factorial_Iterative;
```

Analysis

- Consider: $n! = 1 * 2 * \ldots * (n-1) * n$

so: $(n-1)! = 1 * 2 * \ldots * (n-1) \Rightarrow$

$n! = (n-1)! * n$

We have defined the **!** function in terms of itself

Recursion

- A function that is defined in terms of itself is called *self-referential*, or ***recursive***.

- Recursive functions are designed in a 3-step process:

  1. Identify a **base case**: an instance of the problem whose solution is ***trivial***

     Example: The factorial function has **two base cases**:
     if n = 0 : n! = 1
     if n = 1 : n! = 1

Induction Step

2. Identify an **induction step**: a means of solving the non-trivial instances of the problem using one or more "smaller" instances of the problem.

   Example: In the factorial problem, we solve the "big" problem using a "smaller" version of the problem:

   $n! = (n-1)! * n$

3. Form an algorithm from the base case and induction step.

Algorithm

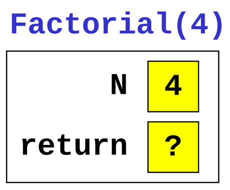
```
-- Factorial(N)

0. Receive N

1. if N > 1
      return Factorial(N-1) * N
   else
      return 1
```

Ada Code

```ada
function Factorial (N : Natural) return
                                  Positive is

  begin -- factorial

    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

Behavior

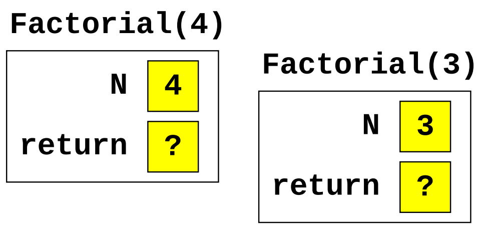
The function starts executing, with N = 4.



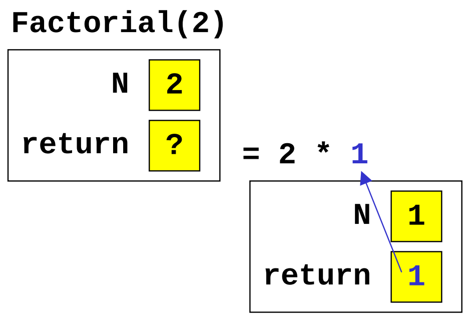
```
function Factorial (N : Natural)
   return Positive is

   begin – factorial

    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

Behavior

Factorial(1) terminates, returning 1 to Factorial(2).

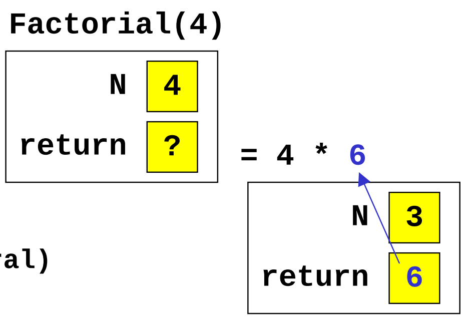


```
function Factorial (N : Natural)
   return Positive is

   begin – factorial

    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```



Behavior

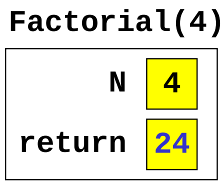
Factorial(3) terminates, returning 6 to Factorial(4):



```
function Factorial (N : Natural)
   return Positive is

   begin – factorial

    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

Behavior

Factorial(4) terminates, returning 24 to its caller.



```
function Factorial (N : Natural)
   return Positive is

   begin – factorial

    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

- If we time the for-loop version and the recursive version, the for-loop version will usually win, because the overhead of a function call is far more time-consuming than the time to execute a loop.

For example the exponentiation problem:

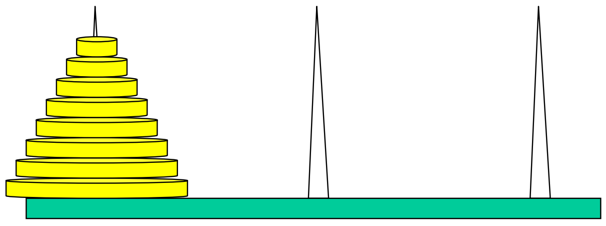
Given two values *x* and *n*, compute x^n.

Example: $3^3 = 27$

- However, there are problems where the recursive solution is more efficient than a corresponding loop-based solution.

A Legend

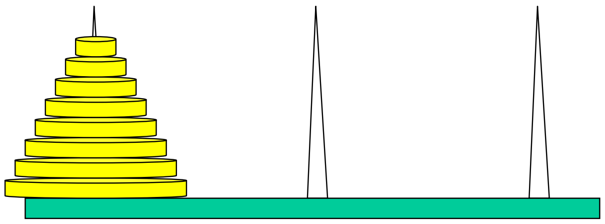
Legend has it that there were three diamond needles set into the floor of the temple of Brahma in Hanoi.



Stacked upon the leftmost needle were 64 golden disks, each a different size, stacked in concentric order:

A Legend (*Ct'd*)

The priests were to transfer the disks from the first needle to the second needle, using the third as necessary.

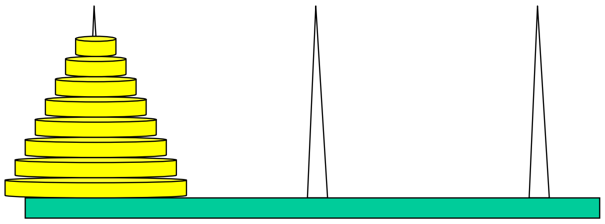


But they could *only move one disk at a time*, and could *never put a larger disk on top of a smaller one*.

When they completed this task, **the world would end**!

Our Problem

Today's problem is to study/write a program that generates the instructions for the priests to follow in moving the disks.



While quite difficult to solve iteratively, this problem has a simple and elegant *recursive* solution.

Example

- Consider six disks instead of 64
- Suppose the problem is to move the stack of six disks from needle 1 to needle 2.
  - Part of the solution will be to move the bottom disk from needle 1 to needle 2, as a single move.
  - Before we can do that, we need to move the five disks on top of it out of the way.
  - After we have moved the large disk, we then need to move the five disks back on top of it to complete the solution.

Example

- We have the following process:
  - Move the top five disks to needle 3
  - Move the disk on needle 1 to needle 2
  - Move the disks on needle 3 to needle 2
- Notice that part of solving the six disk problem, is to solve the five disk problem (with a different destination needle). Here is where recursion comes in.

Algorithm

-
```
hanoi(from,to,other,number)
    -- move the top number disks
    -- from needle from to needle to
if number=1 then
    move the top disk from needle from to needle to
else
    hanoi(from,other,to, number-1)
    hanoi(from,to,other, 1)
    hanoi(other,to, from, number-1)
end
```

Analysis

Let's see how many moves it takes to solve this problem, as a function of *n*, the number of disks to be moved.

n	Number of disk-moves required
1	1
2	3
3	7
4	15
5	31
…	
i	2^i-1
64	2^{64}-1 (a big number)

PRS2 -- Recursion

Assume that the user enters:

Hi!(end of line)

1. Displays Hi! on the same line
2. Displays Hi! on the next line
3. Displays !iH on the same line
4. Displays !iH on the next line