Departamento de Informática
Universidad de Valladolid
Campus de Segovia

TOPIC 1:
RECURSION

INDEX

- Direct recursion. Definition
- How direct recursion works
- Some examples
- Mutual recursion
- Recursion vs iteration

DIRECT RECURSION. DEFINITION

- It is a mathematical way of defining a problem in terms of a smaller version of itself.
- As a programming technic it involves procedures or function calling itself and implies a different way of looking at the repetitive actions.
- However this definition is not enough to understand it depthly since some aspects related to how it works need a more detailed description.
  - How can we define the solution in terms of a smaller solution?
  - How is the size of the solution being disminished at each recursive call?
  - How does the recursive algorithm detect that the final solution is reached?

HOW DIRECT RECURSION WORKS.
THE FACTORIAL OF AN POSITIVE INTEGER

- The factorial of an integer n! may be defined as:

$$n! = n\,(n-1)\,(n-2)........1$$

- According to that, (n-1)! is defined as

$$(n-1)! = (n-1)\,(n-2).......1$$

- Therefore, a recursive definition of that function may be:

$$n! = n\,(n-1)! \quad \text{si } n>0$$
$$0! = 1 \quad \text{si } n=0$$

HOW DIRECT RECURSION WORKS.
THE FACTORIAL FUNCTION

Fac(4)=4*Fac(3)

Fac(4)=4*(3*Fac(2))

Fac(4)=4*(3*(2*Fac(1)))

Fac(4)=4*(3*(2*(1*Fac(0))))

Fac(4)=4*(3*(2*(1*1)))

Fac(4)=4*(3*(2*1))

Fac(4)=4*(3*2)

Fac(4)=4*6=24

PASCAL IMPLEMENTATION

```
FUNCTION Fac(number:integer):integer;
{Prec. number≥0}
Begin
  if number = 0 then
      Fac := 1
  else
      Fac := number * Fac(number-1)
End; {Fac}
```

- The call is within the function body.

HOW DIRECT RECURSION WORKS.

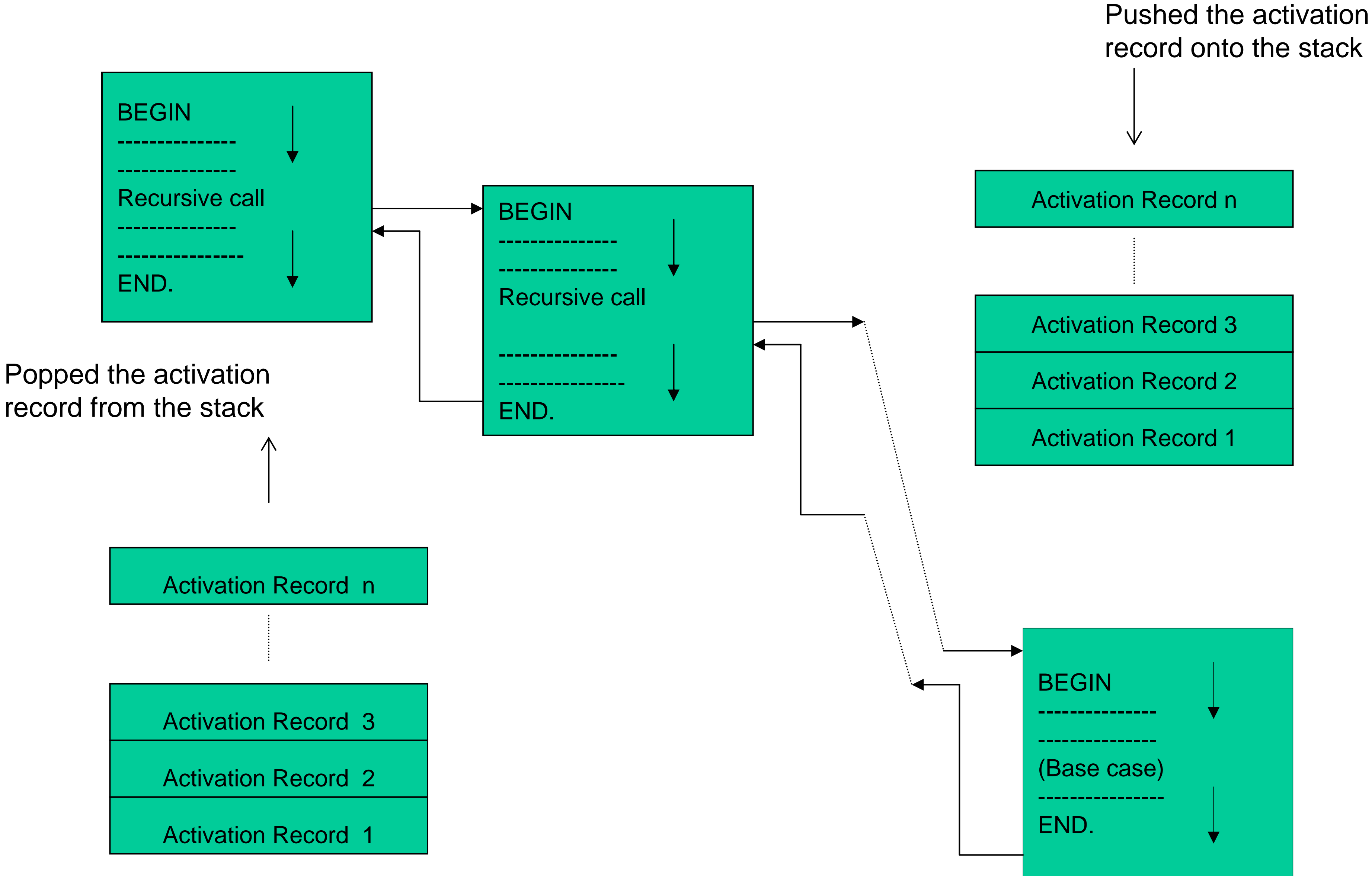
Therefore a recursive algorithm need:

- A **General case** that provides a definition of the problem in terms of itself.
- A **base case** defined as which that stop the recursion process when the final solution is reached.
- Any recursive algorithm must have, at least ,one base case.

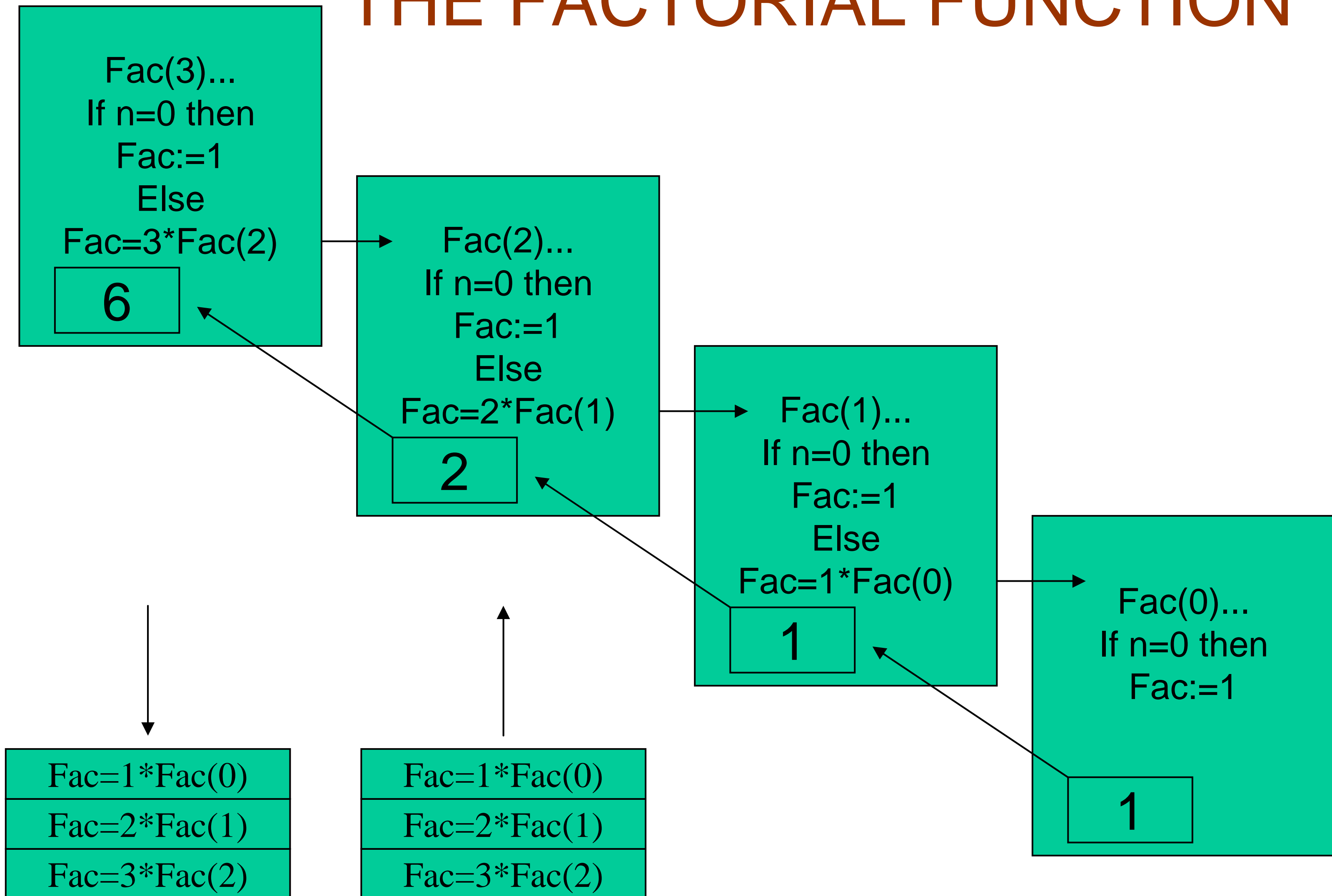
HOW DIRECT RECURSION WORKS

- Running a recursive program:
  - Once a recursive call is invoked the current values of the local variables and formal parameters are stored in a work space called **activation record**.
  - Then the activation record is stored and a new smaller version of the recursive program is active. This process is repeated until a case base is reached. Then no more recursive calls are invoked and the final solution is built using the partial solution. The order of activation of these ones follow the **Last-in First–out rule** (LiFo).
  - The structure that supports such rule is an stack called **run time stack**.

HOW DIRECT RECURSION WORKS



HOW DIRECT RECURSION WORKS. THE FACTORIAL FUNCTION



HOW DIRECT RECURSION WORKS. THE FACTORIAL FUNCTION

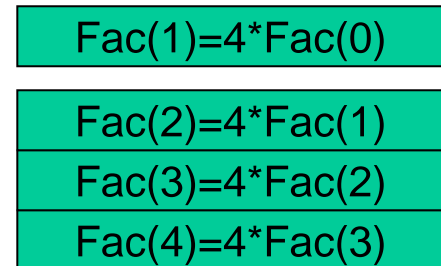
Fac(4)=4*Fac(3)

Fac(4)=4*(3*Fac(2))

Fac(4)=4*(3*(2*Fac(1)))

Fac(4)=4*(3*(2*(1*Fac(0))))

Pushed onto the stack



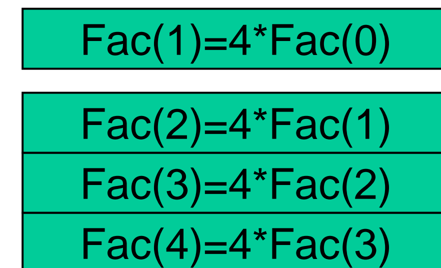
Fac(4)=4*(3*(2*(1*1)))

Fac(4)=4*(3*(2*1))

Fac(4)=4*(3*2)

Fac(4)=4*6=24

Popped from the stack



THE FIBONACCI FUNCTION

- Fibonacci function definition:

  - f(n)= 1 si n=1,0
  - f(n)= f(n-1)+f(n-2) si n>1

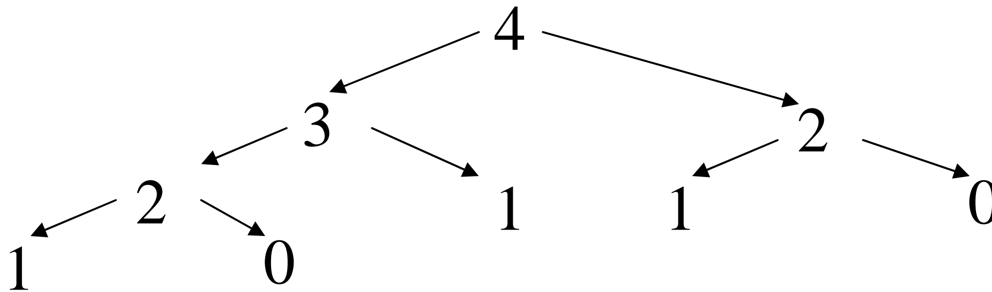
- PascaL implentation:

```
FUNCTION fib(number:integer):integer;
{Prec. number≥0}
Begin
    if (number=0) or (number=1) then
            fib:=1
    else
            fib:=fib(number-1)+fib(number-2)
End; {fibonacci}
```

RUN-TIME DEFINITION OF NEW BASE CASES

- Sometimes a recursive call with the same arguments is repeated as the recursive program is running.

- Función de Fibonacci: Fib(4)



- In order to improve the efficiency of this algorithm a run-time table can be implemented providing a memory space where a new recursive call can be stored as a base case.

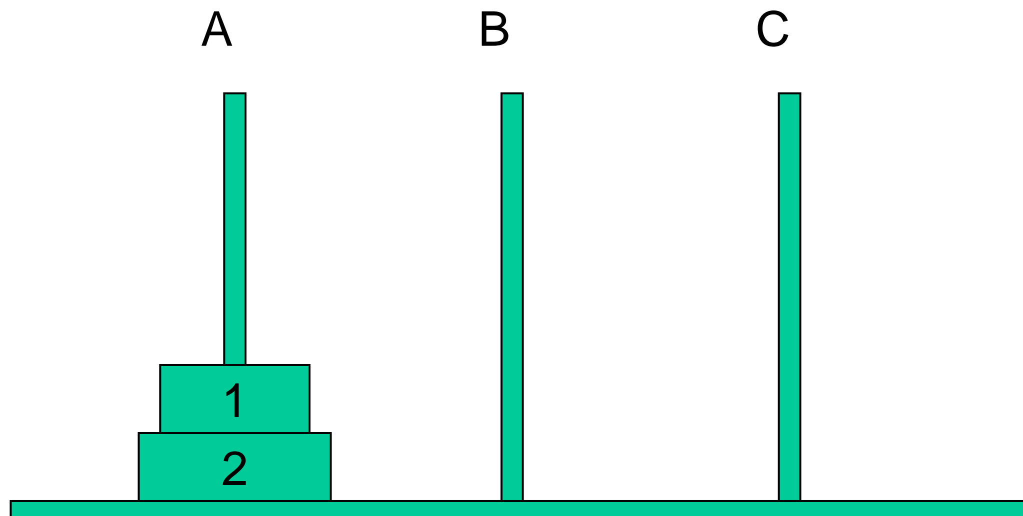
RUN-TIME TABLE IMPLEMENTATION

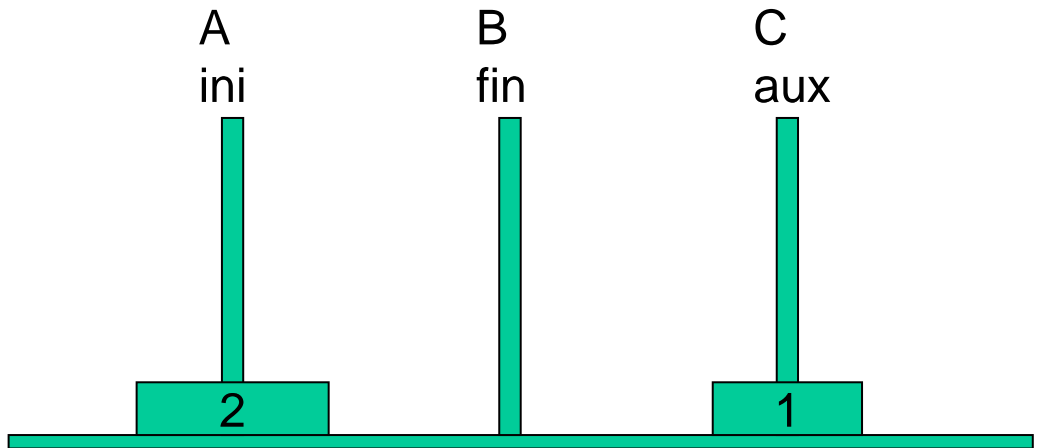
```
type
  tdomain =0..20
  tregister= record
            defined: boolean;
             result: integer
         End;
  tTabla=array [tdomain] of tregister;
 var
 tablafib:tTabla; i:tdomain;
 Function Fib(n: tdomain; var
    t:tTabla):integer;
  Begin
     if not t[n].defined then Begin
        t[n].defined:=true;
        t[n].result:=Fib(n-1,t)+Fib(n-2,t);
      End {if};
   Fib:=t[n].result
 End; {Fibonacci}

 Begin
 tablafib[0].defined:=true;
    tablafib[0].result:=1;
 tablafib[1].defined:=true;
    tablafib[1].result:=1;
 for i:=2 to 20 do
    tablafib[i].defined:=false;
.......
```

THE HANOI TOWERS PROBLEM

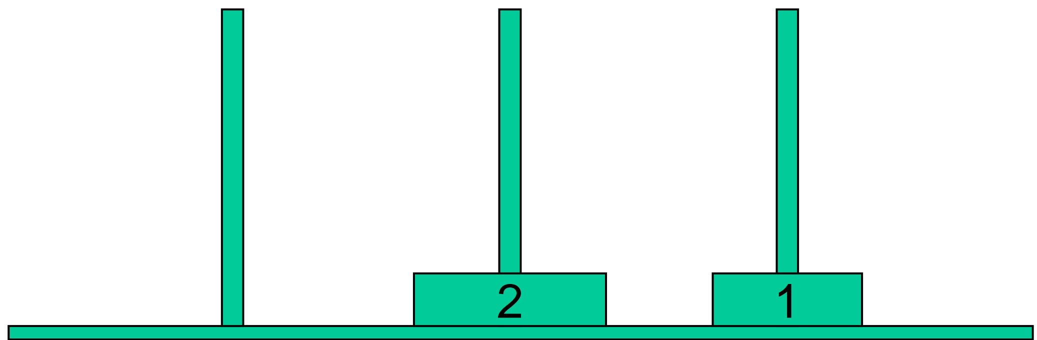
– Move all discs from tower A to B, but moving one by one and only considering allowed to stack a smaller disc over a greater one.





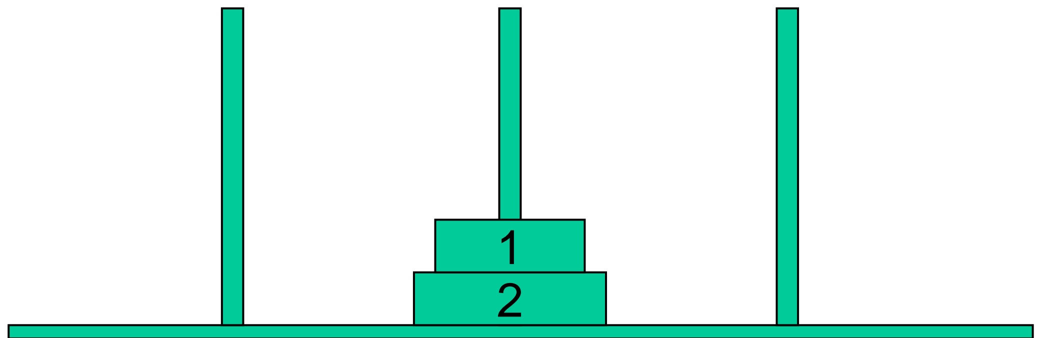
Move one disc
from A to C

ini-aux



Move one disc
From A to B

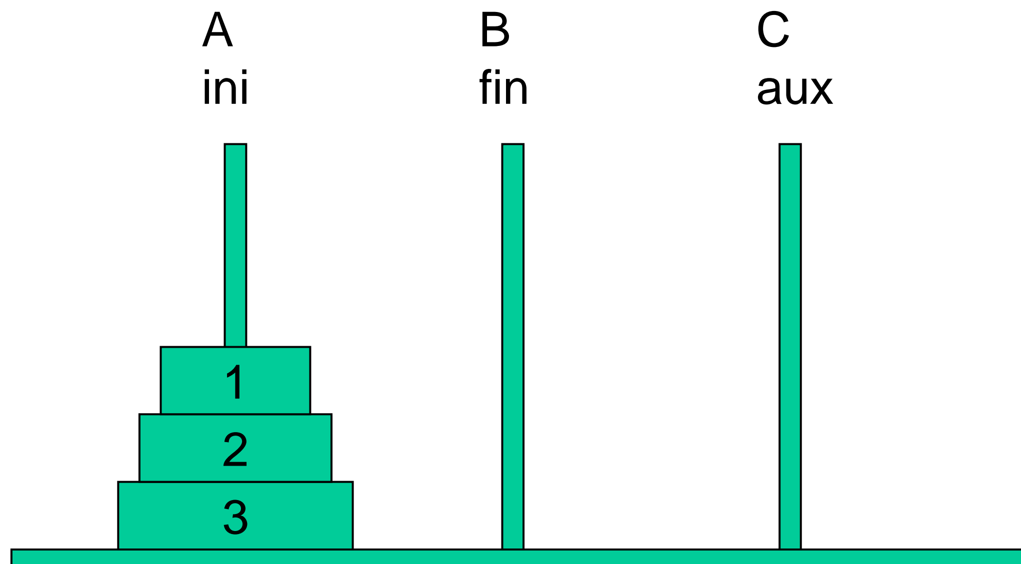
ini-fin



Move one disc
From C to B

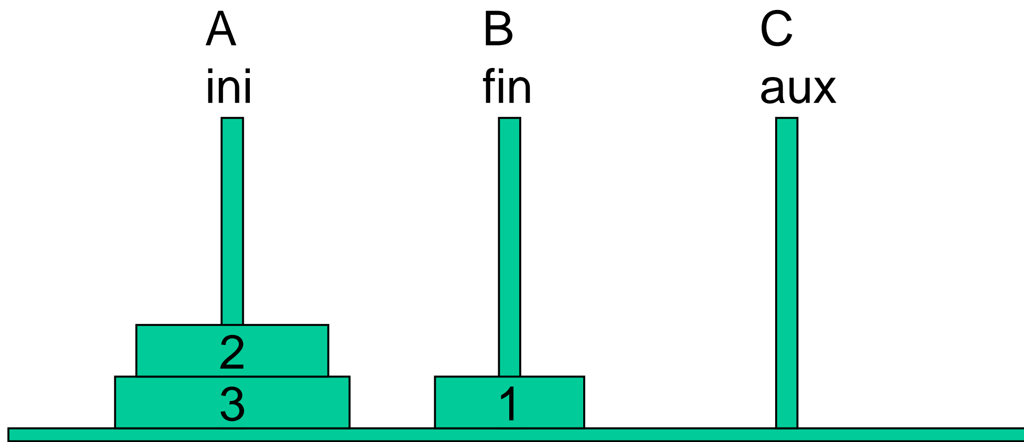
aux-fin

THE THREE DISC HANOI TOWERS PROBLEM

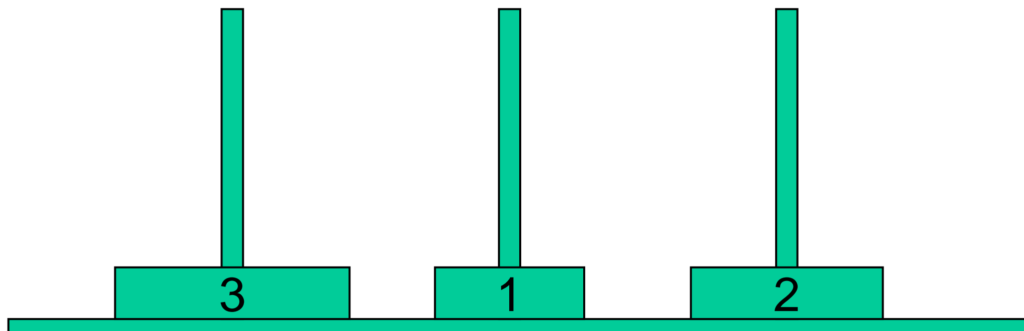


Procedure arguments:
(n, ini, fin, aux)

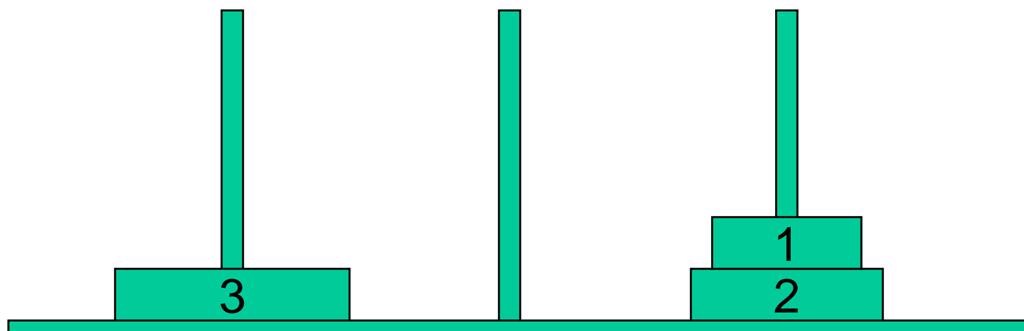
Move the two upper discs from A to B



Ini-fin



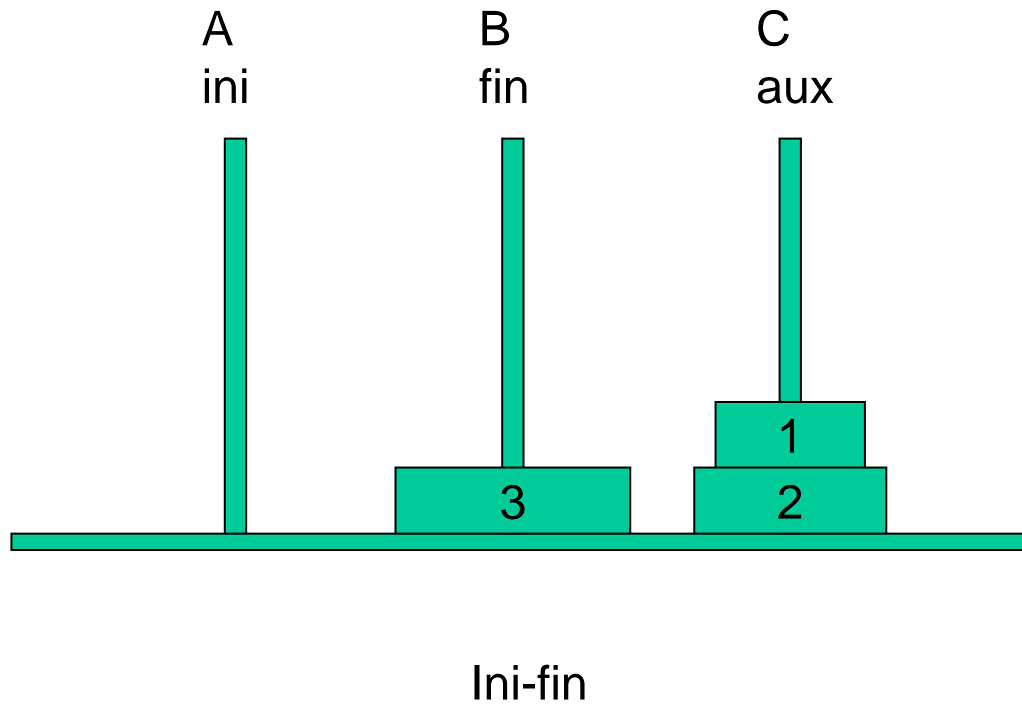
Ini-aux

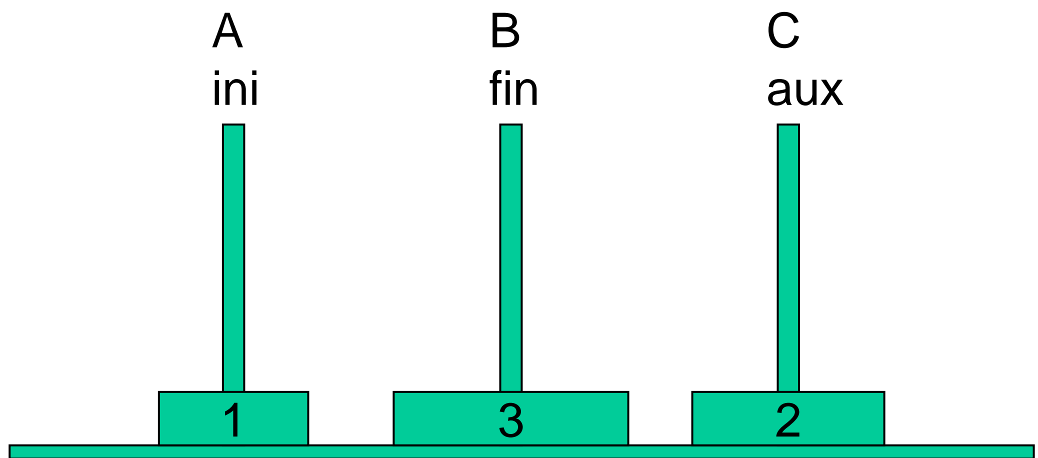


fin-aux

(n-1,ini, aux, fin)

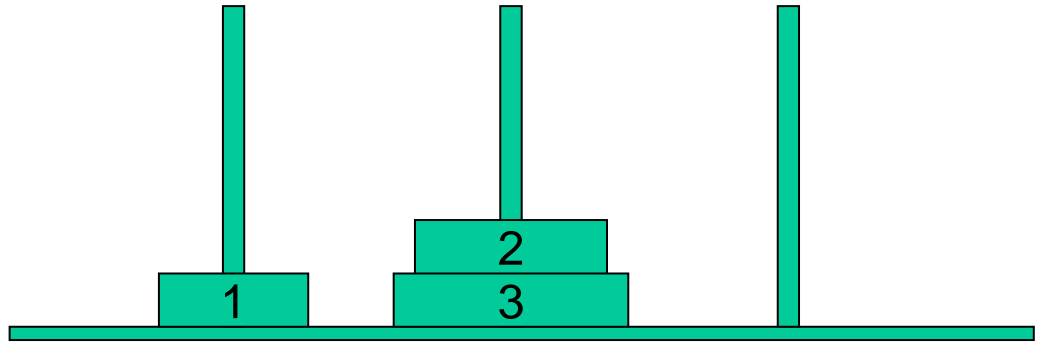
Move the third disc from A to B



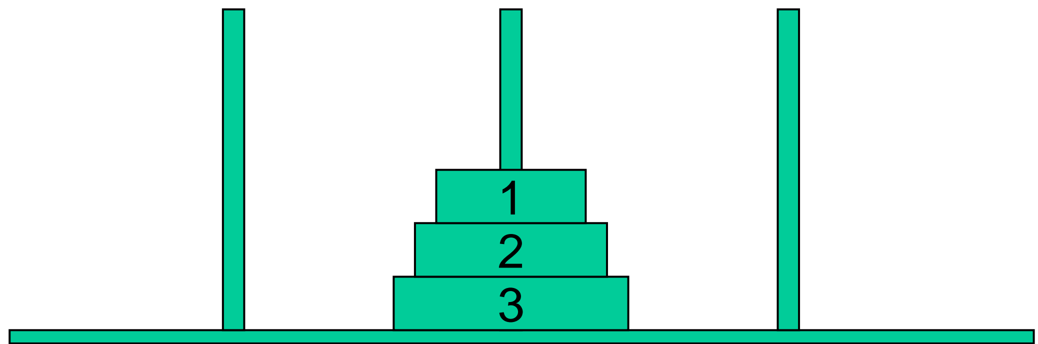


Move two discs from C to B

aux-ini



aux-fin



ini-fin

(n-1,aux, fin, ini)

Algorithm to move three discs from A to B

1. Move two discs from A to C:
    a. Move one disc from A to B
    b. Move one disc from A to C
    c. Move one disc from B to C

2. Move one disc from A to B

3. Move two discs from C to B:
    a. Move one disc from C to A
    b. Move one disc from C to B
    c. Move one disc from A to B

MOVING “N” DISCS FROM A TO B. RECURSIVE DEFINITION:

1. Move n-1 disc from A to C

2. Move one disc from A to B

3. Move n-1 discs from C to B

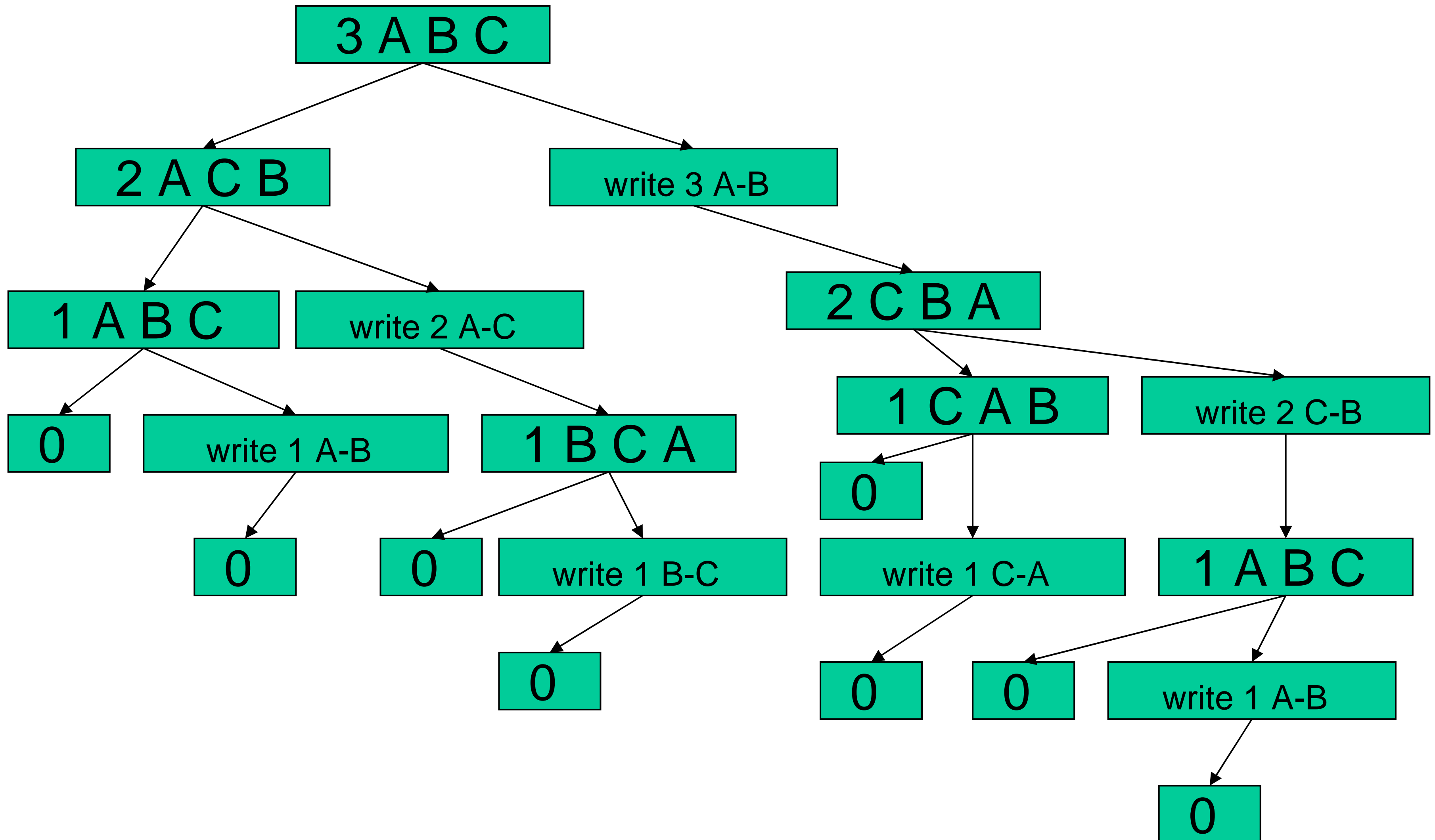
Base case if n=0

PASCAL IMPLEMENTATION

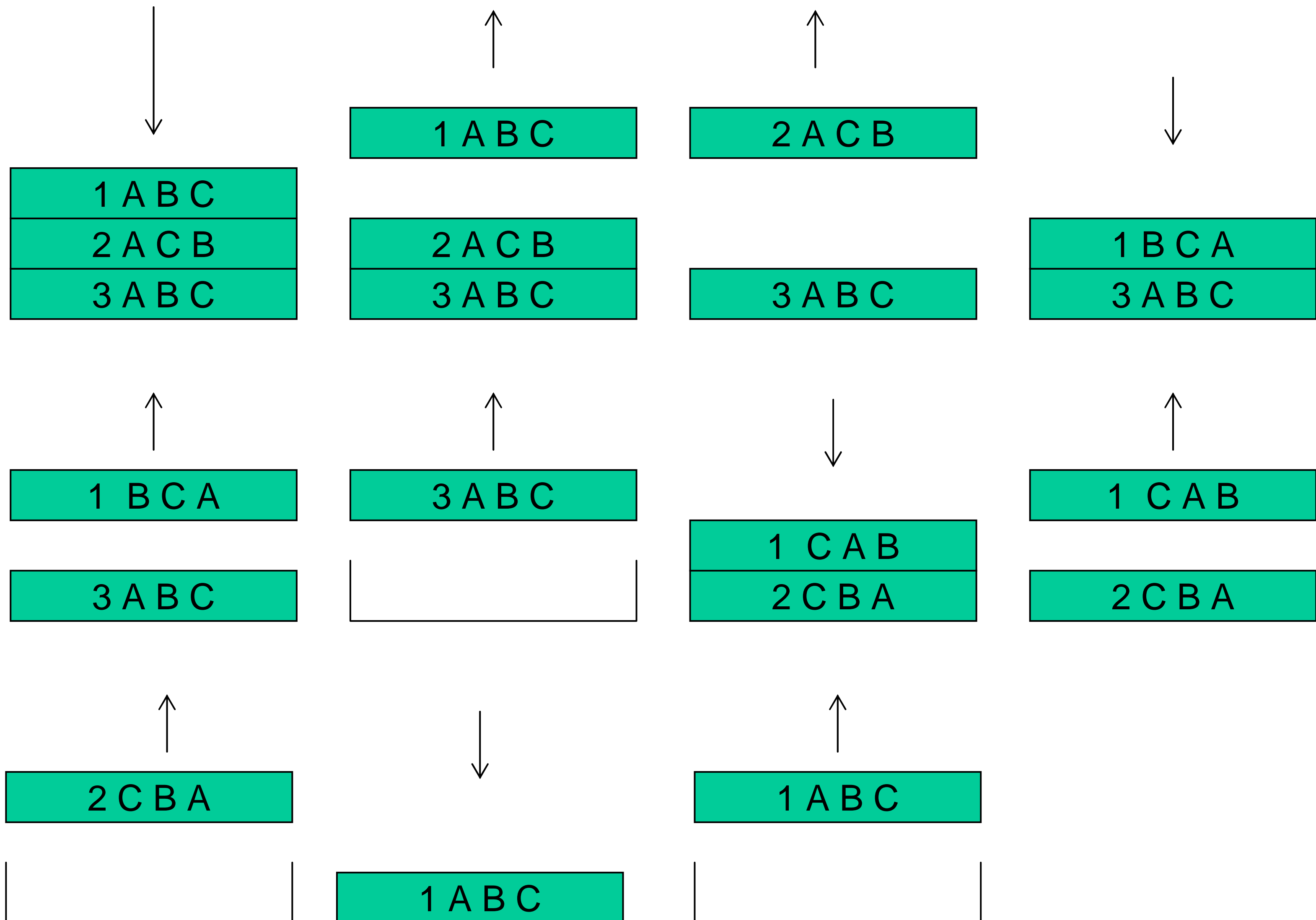
```
PROCEDURE Movingdisc(n:integer; ini,fin,aux:char);
{Prec. n≥0}

Begin
    if n>0 then begin
        Movingdisc(n-1,ini,aux,fin);
        writeln('move disc', n:3,'from', ini,'to', fin);
        Movingdisc(n-1,aux,fin,ini)
    end {if}
End; {Movingdisc}
```

A SEQUENTIAL DESCRIPTION OF THE ALGORITHM FOR N=3



A RUN-TIME STACK DESCRIPTION



MUTUAL RECURSION

- **Mutual recursion** is a form of recursion where two mathematical or computational functions are defined in terms of each other.

- In order to check and compile a routine call, a compiler must know the type of the routine, the number of parameters and so on. Since routines must be defined in some order at least one routine must be compiled before its definition is seen.

- Pascal programming language uses separate **forward** definitions of routine headers to give sufficient information to compile the recursive call and **body** definitions to contain these ones.

PASCAL IMPLEMENTATION OF THE MUTUAL RECURSION

```
PROCEDURE Second (arguments); forward;
PROCEDURE First (arguments);
........
Begin {First}
.........
Second(arguments)
.........
End; {First}
PROCEDURE Second (arguments);
........
Begin {Second}
.........
First(arguments)
.........
End; {Second}
```

MUTUAL RECURSION.
CHECKING THE PARITY OF AN INTEGER

- Implement a pascal program to check the parity of a positive integer using mutual recursion.
- If parity is the quality of being either odd or even, the algorithm can be implemented defining two procedure mutually dependent:

```
Function even(n:integer):boolean;
Function odd(n:integer):boolean;
```

```
Program parity;
uses
    crt;
Var
    n:integer;
PROCEDURE odd(n:integer); forward
PROCEDURE even(n:integer);
    Begin
        if n=0 then
            writeln('the number is even')
        else
            odd(n-1)
    End;{even}

PROCEDURE odd(n:integer);
    Begin
        if n=0 then
            writeln('the number is odd')
        else
            even(n-1)
    End;{odd}

Begin {parity}
    clrscr;
    writeln('enter a positive integer number');
    readln(number);
    even(number);
    readln {pause}
End; {parity}
```

RECURSION vs ITERATION

- Since a recursive program involves repetition then it may be implemented as well by means of an iteration and viceverse.

- But, how to decide which of the two have to be used? In this case is necessary to take into account some aspect such as eficiency and legibility.