Chapter 1(part 2): Recursion

1. Notion of the recursion:

➤ In programming, ***recursion*** is a method that allows a sub-program (procedure or function) **of calls itself**.

➤ It is in <u>their part body</u> (instructions) we find a call to the procedure (or function) itself.

Example: write an algorithm (function) to calculate the factorial of a given integer N?

a) <u>Solution iterative (classic) :</u>

N !=1*2* 3*4*5*… .... *(N-1)*N.

Function fact (N: integer): integer;

R, i: integer;

Begin

    R ← 1;

    For i = 1 **to** N **do**

      R ← R* i ;

    End For

    Return (R) ;

END ;

b) <u>Solution recursive:</u>

N !=N*(N-1) !

  = N*(N-1)*(N-2)!

  =N*(N-1)*(N-2)* ....* 0!

➤ the function factorial "***fact***" is defined as follows :

    ✓ If N = 0 :   fact (0) = 1.
    ✓ If N < > 0:   fact (N)=N*fact(N-1).

➤ In algorithmic, the function ***fact*** is defined as follow :

Function fact (N: integer): integer;

  R: integer ;

<u>**Begin**</u>

  If (N = 0) then
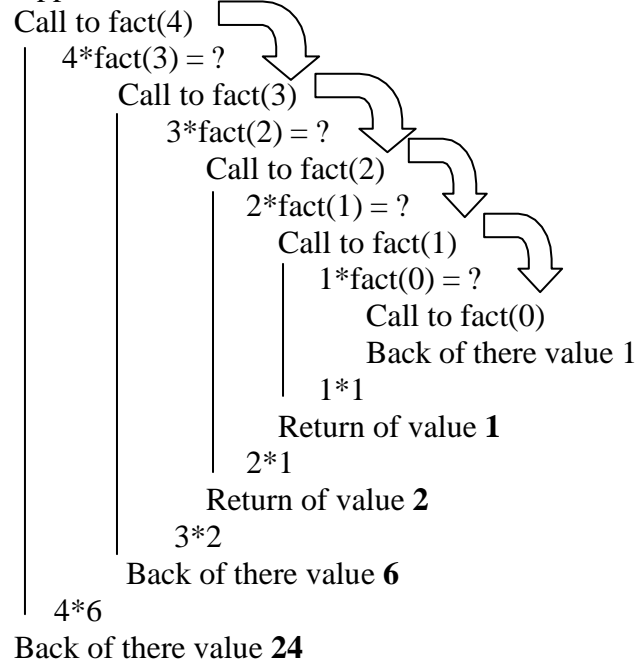
  R ← 1;

  Else

      R ← N * fact (N - 1);

  End if

  Return (R) ;

<u>**END;**</u>

Recursion call

Application for 4! :
Call to fact(4)
  4*fact(3) = ?
    Call to fact(3)
      3*fact(2) = ?
        Call to fact(2)
          2*fact(1) = ?
            Call to fact(1)
              1*fact(0) = ?
                Call to fact(0)
                Back of there value 1
            1*1
            Return of value **1**
          2*1
          Return of value **2**
        3*2
        Back of there value **6**
  4*6
Back of there value **24**

2. Types of recursion:

➢ We distinguished usually two types of recursion :
- ✓ **simple** recursion
- ✓ **crossed** recursion

The simple recursion:

➢ This is when a subroutine (function or procedure) calls itself. This is indeed the general case of recursion, as we have already seen in the previous example with the function **fact**.

<u>Syntax :</u>

Procedure P
Begin
   …
   Call to P;
   …
END ;

<u>Example</u> : calculation of the sum of the first **n** positive integers.

Sum (n) = 1 + 2 + 3 + … + (n-1) + N

Function Sum (N : integer) : integer;
S: integer;
Begin
  If (n = 1) then
      $S \leftarrow 1$;
  Else
     $S \leftarrow$ Sum (n-1) + n ;
  End if
  Return (S) ;
END;

The cross recursion:

➢ We call cross recursion the do that **two procedures** P1 And P2 call **each other**, i.e. when **P1** executes, it calls **P2** , and when **P2** executes, it calls to **P1** .

<u>Syntax:</u>

Procedure P1
Begin
  …
  Call to P2 ;
  …
END ;
Procedure P2
Begin
  …
  Call to P1 ;
  …
END ;

<u>Example :</u>
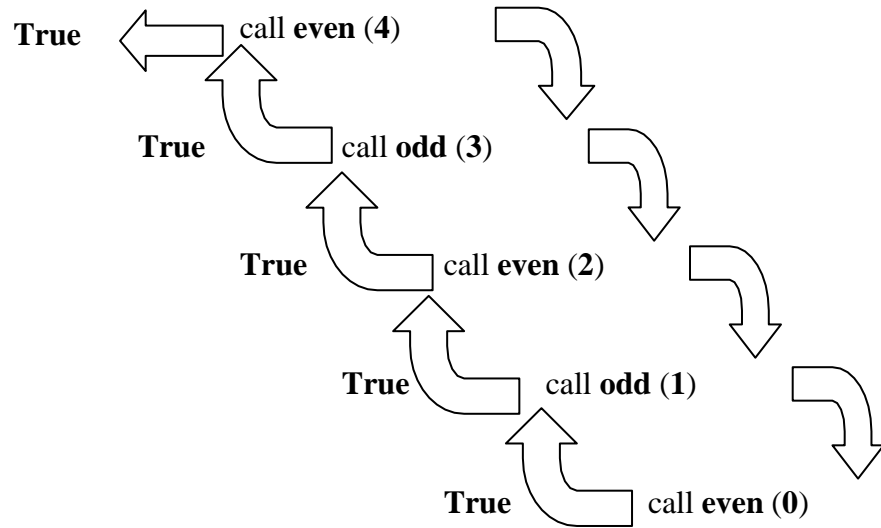➢ A positive integer **n** can be either :
Even → n = 2*k
Odd → n = 2*k+1
➢ If we considered two functions **Even** (n) And **Odd** (n) has logicalvalues (**boolean**) then we will have :
    If **Even** (n) = true   then **Odd** (n) = false
    If **Odd** (n) = true   then **Even** (n) = false

Function even (n : integer): boolean ;

R: boolean;

<u>Begin</u>

  if (n = 0) then

       R ← true ;

  else

      R ← odd (n-1) ;

  End if

  return (R) ;

<u>End ;</u>

Function odd (n : integer): boolean;

 R: boolean;

 <u>Begin</u>

  if (n = 0) then

       R ← false ;

  else

      R ← even (n-1) ;

  End if

  return (R) ;

<u>End ;</u>

Application: we want to check **even (4):**

True   call **even** (**4**)

True   call **odd** (**3**)

True   call **even** (**2**)

True   call **odd** (**1**)

True   call **even** (**0**)

---

3. Rules of design :

a) <u>First rule :</u>

➢ Seek to break down the problem into several below problems of the same kind but lower size.

b) <u>Second rule :</u>

➢ Any recursive algorithm must distinguish several cases, at least one of which born must not include recursive call (*stop Condition Or ending*). It is the **trivial** case, otherwise risk of loop infinitely.

Condition of termination:

✓ Non-recursive cases of a recursive algorithm are called *base cases*.
✓ The conditions that the data must satisfy in these base cases are called **conditions of ending**.

<u>Example :</u>

> **Function** fact (N : integer):integer ;
> R: integer ;
> **<u>Begin</u>**
> Return (N * fact (N - 1)) ;
> **<u>END;</u>**

Call to fact(4)

  4*fact(3) = ?

    Apple at fact(3)

      3*fact(2) = ?

        Call to fact(2)

          2*fact(1) = ?

            Call has fact(1)

              1*fact(0) = ?

                Call to fact(0)

                  0*fact(- 1) = ?

                    …

Without **conditions of ending**?!

<u>Solution</u> *:*

     Function fact (N: integer): Integer;

     R : integer;

     <u>Begin</u>

     If (N = 0) then   *Base case*

     R ← 1;

     Else

          R ← N * fact (N - 1);

     End if

      Return (R) ;

     <u>END;</u>

<u>Noticed :</u>

➢ Since a recursive function calls itself, it is imperative that we provide for a condition stop to the recursion, otherwise the program does **never stop**.

➢ We must always test the stopping condition first, and then, if the condition is not verified, we start a recursive call.

c) **<u>Third rule :</u>**

➢ All call recursive must to do with data closer to the data that satisfying the termination condition.

<u>Benefits of the recursion:</u>

➢ Simplify the writing of programs, because the calculations to be performed are not explicitly defined.

➢ Facilitate the task of the programmer who will no longer have to specify the number of iterations of the same action, neither have to manage the values of the different variables used.