

Subscribe to DeepL Pro to edit this document.  
Visit [www.DeepL.com/pro](https://www.deepl.com/pro?cta=edit-document) for more information.

**Batna2 University**

**IT Department**

**Academic year 2022-2023**

**Algorithms and data structure 2**

**Course outline :**

**Chapter 1:** Subprograms: Functions and Procedures

**Chapter 2:** Files

**Chapter 3 :**Linked lists

Ms Malika Bachir - first year, second semester 2022-2023. Batna

1. **Introduction**

In programming (algorithmics), as soon as you start writing large programs, it becomes difficult to have an overall view of how they work and to track down errors. In some programs, for example, you'll find a repetition of the same problem over and over again.

For example, in the same algorithm we were asked to calculate :

1. The maximum factorial of an array
2. The maximum factorial of a matrix with n rows and m columns
3. The maximum factorial of a matrix with m rows and n columns
4. nm
5. mn

What can be done to simplify the execution, correction, maintenance, etc. of this program?⇒ ***Break*** the problem down into sub-problems and find a solution for each, then ***combine*** them into a single program (algorithm). In Algorithmics (programming), each partial solution gives rise to a sub-algorithm (sub-program) which will be part of a complete algorithm before it can be executed.

1. **Definitions**

A sub-algorithm is a block within an algorithm. It is **declared** in the **header** (before the start of the algorithm) and then **called** in the body of the algorithm.

As the sub-algorithm is a block in its own right, it has a header, a series of processes and a results management system, just like the algorithm that contains it.

A sub-algorithm is declared in a general way, i.e. it can be called several times with different values thanks to arguments. These arguments, although optional, are called parameters and are clearly declared, if necessary, in the subalgorithm header.

A parameter is a value in the main block that the sub-algorithm needs to execute the sequence of actions it is instructed to perform with real data. There are two types of parameter:

* **Formal parameters** define the **number and type of** values that the sub-algorithm must receive in order to run successfully. Formal parameters are declared when the sub-algorithm is declared.
* **Effective parameters** are **real values** (constants or variables) received by the sub-algorithm during execution of the main block. They are defined independently each time the sub-algorithm is called in the main algorithm.

A sub-algorithm (procedure or function) is **executed by a call instruction**. Applying this instruction generates a jump to the called sub-algorithm. Termination of this sub-algorithm restarts the instruction sequence interrupted by the call.

1. **Local and global variables**

A sub-algorithm uses the variables declared in the algorithm (called global variables). It can also have its own variables (called local variables) declared in the space reserved for it; but these can only be used in this sub-algorithm and nowhere else, as their scope (visibility) is limited to the block containing them. The space for these local variables is reserved only when the sub-algorithm is called, and is freed at the end of execution.

1. **Sub-algorithm types**

A sub-algorithm can be either a function or a procedure.

A function is a sub-algorithm which, given data, **calculates and returns** a single result to the algorithm, whereas a procedure generally **displays** the requested result(s).

* 1. **Function**

A function is a block of instructions that returns one and only one result value to the calling algorithm. A function never displays the answer on the screen, as it simply returns it to the calling algorithm.

* **Declaring a function :**

|  |
| --- |
| **Function\_Name(Parameter\_Name : Parameter\_Type;......): function\_type**  **Optional**  Declaration  Variable\_name: Variable\_type{Local **variables}**  ...  **Start**  ...  Instructions; Body of the procedure  ...  **Function\_Name**← Result/ or return (Result){Required}  **FinFonctio** |

Since the main purpose of a function is to return a value, it is necessary to specify **the type of the function**, which is actually **the type of this value** (the result).

* **Calling a function :**

A function call is an assignment expression so that the result is retrieved in a global variable: **Global\_variable\_name← Function\_name (parameters) ;**

**Example**: Here's an algorithm using a function that calculates a sum of 100 numbers.

|  |
| --- |
| Test **algorithm**  **Variable**  I, Som: integer (**global variable**)  Sum **function**: integer  Variable  S: integer (**local variables**)  **Start** /\*Function start\*/  S← 0  For I from 1 to 100 Make  S← S + I  EndFor  Sum← S (or return S)  **End** /\*End Function \*/  **Start** /\*Start of algorithm\*/  **Som← Somme (function call)**  Write ('The sum of the ', N, 'first numbers is', Som)  **End** /\*End of algorithm\*/ |

**Note**: A function can call other sub-algorithms, provided they have been defined before it or declared in its header.

* 1. **Procedure**

A procedure is a block of instructions named and declared in the header of the algorithm and called in its body whenever the programmer needs it. A procedure is exactly the same as a function, except that the latter can return zero or more results.

* **Reporting a procedure :**

|  |
| --- |
| **Procedure\_Name** (Parameter\_Name : Parameter\_Type;......)  **Facultatit**  Declaration  Variable\_name: Variable\_type {Local **variables}**  ...  **Start**  ...  Instructions; Body of the procedure  ...  **End** |

* **Calling a procedure :**

A procedure can be called up by specifying its name and, if required, its parameters at the desired moment; this triggers the execution of the procedure's instructions.

**Example**: The previous algorithm, which calculates a sum of N numbers, can use a procedure instead of a function.

|  |
| --- |
| Test **algorithm**  **Variable**  I, S :integer**(global variable)**  Procedure Somme  Start /\*Start Procedure\*/  S← 0  For I from 1 to 100 Make  S← S + i  EndFor  Write ('The sum of the first 100 numbers is', S)  End /\*End of Procedure\*/  Start /\*Start of algorithm\*/  **Sum (Procedure call)**  End /\*End of algorithm\*/ |

1. **Parameter switching**

There are two ways of passing parameters: by **value** and by **variable** (also known as by reference or by address).

* 1. **Passing parameters by value**
* This is the default transmission mode, copying the value of the effective parameters into local variables derived from the formal parameters of the called procedure or function.
* In this mode, the contents of effective parameters cannot be modified by function or procedure instructions, as we are not working directly with the variable, but on a copy.
* At the end of the sub-algorithm's execution, the variable retains its initial value. In this case, the parameters are used as data.

**Syntax** :

Procedurename\_procedure (param1 :type1 ; param2, param3 :type2)

Function <function\_name> (param1 :type1 ; param2 :type2) : Function\_type

**Example**: Consider the following algorithm.

|  |
| --- |
| pas-val **algorithm**  variable  M: integer  **Procedure** P1 (**number** : integer)  **Start**  If number > 0 Then  number← number\*2  FinSi  Write (number)  **End**  **Start**  Read (M)  P1 (**M**)  Write (M)  **End** |

Running this algorithm for M = 5, we obtain the following execution trace:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| pas-val algorithm   |  |  | | --- | --- | | M | Output | | 5 | **5** | | P1 procedure   |  |  | | --- | --- | | number | Output | | 5  10 | 10 | |
| * At the start of the algorithm, variable M is given the value 5 * The algorithm then calls the P1 procedure with the effective parameter M * Execution of procedure P1 results in the value 10 * On returning to the algorithm (at the call level), all that remains is to display the global variable M with its initial value of 5. So, the algorithm displays on screen in order: 10 then 5, i.e. M hasn't changed value (M hasn't received the result). | |

* **We conclude that passing by value does not change the content of the effective parameter.**
  1. **Passing parameters by variable**

Here, it's not just a question of using the value of the variable, but also its location in memory (hence the expression "by address"). In effect, the formal parameter replaces the actual parameter during the execution of the sub-program, and on exit transmits its new value. Alternatively, the **address of the** effective parameter **variable is** transmitted, not its value, so any operation is carried out directly in this variable.

Such parameter passing is done by using the **Var** (or reference) keyword.

**Syntax** :

Procedurename\_procedure (**Var** param1 :type1 ; param2, param3 :type2)

Function <function\_name> (**Var** param1 : type1 ; param2 :type2) : Function\_type

**Note**: Parameters passed by value and by address can coexist within the same sub-algorithm. All you need to do is separate the two pass types with a (**;)**.

**Syntax** :

Procedurename\_procedure (**Var** param1 :type1 ; param2, param3 :type2)

In this case param1 is passed by reference, while the other two are passed by value.

Function <function\_name> (param1 :type1 ; **Var** param2 :type2) : Function\_type

In this case param1 is passed by value while the second is passed by reference.

**Example**:

Let's take the previous algorithm and change the type of parameter passing to address passing

|  |
| --- |
| pas-val **algorithm**  variable  M: integer  **procedureP1** (**Var** number: integer)  Start  If number > 0 Then  number← number\*2  FinSi  Write (number)  End  **Start**  Read (M)  P1 (**M**)  Write (M)  **Fin**. |

Let's run this algorithm again for value (5), so the execution trace will be as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| pas-val algorithm   |  |  | | --- | --- | | @ : 143 |  | | M | Output | | 5  10 | **10** | | P1 procedure   |  |  | | --- | --- | | number | Output | | @ : 143 | 10 | |
| * At the start of the algorithm, variable M is given the value 5 * Next, the algorithm calls the P1 procedure via the effective parameter M (passing by address). * Execution of procedure P1 results in the value 10 * On returning to the algorithm (at call level), all that remains is to display the global variable M, which in this case is 10. So, the algorithm displays on screen in order: 10 then 10, i.e. M has changed its initial value (M has received the result). | |

* **We conclude that passing by address changes the content of the effective parameter.**

1. **Recursivity**
   1. **Definition**

An algorithm (function, procedure) is said to be recursive if its definition (code) contains a call to itself.

An algorithm that is not recursive is said to be iterative.

* 1. **Typical applications**

Various uses (non-exhaustive list) :

* Recursive sequence calculation (numerical, graphical. . . )
* Divide and conquer" calculations: searching, sorting, ...
* Calculation on inductive data structures (lists, trees, . . . )
* In all cases, an iterative version is possible.
  1. **A few classic examples**

|  |
| --- |
| 1. **Factorial**: n! = n.(n - 1)!   Function **fact**(n) : integer  **Start**  **If** n=0 **then**  fact← 1 ( or Return 1)  **Otherwise**  fact← **n\*fact**(n-1) ( or Return n\*fact(n-1) ) **{Recursive call}**  **Fsi**  **FFunction** |

|  |
| --- |
| 1. **Fibonacci:** Fibo(n) = Fibo(n - 1) + Fibo(n - 2)   Function **fibo**(n) : integer  **Start**  **If** n=0 **then**  fibo← 1 (or Return 1 )  **Otherwise**  **If** n=1 **then**  fibo← 1 (Return 1)  **Otherwise**  fibo← **fibo**(n-1)**+fibo**(n-2) (or Return fibo(n-1)+fibo(n-2) ) **{Recursive call}**  **Fsi**  **Fsi**  **FFunction** |

|  |
| --- |
| 1. **What does sum(5,0) calculate?**   Function **sum**(n,r) :integer  **Start**  **If** n = 1 **then**  Sum← 1( orReturn r + 1)  **Otherwise**  Sum← **sum**(n -1 , r + n ) ( or Return sum (n -1 , r + n ))**{Recursive call}**  **Fsi**  **FFunction** |

1. **C functions** 
   1. **Definition**

**function\_return\_type**(list-params)

{

list-declarations (optional)

list\_instructions

return (**of type\_of\_return**)

}

The instruction list includes at least onereturn instruction.

* 1. **Call**

function\_name(expression-list)

* 1. **Example**

***Defining a function***

int max ( int a , int b )

{

int m;

i f (a>b)

m=a ;

else

m=b ;

return (m) ;

}

***Call***

T = max(3 ,45);

Using this function to display the maximum of an array, we write the following program:

|  |
| --- |
| #include<stdio.h>  int max ( int a , int b )  {  int m;  if (a>b)  m=a ;  else  m=b ;  return (m) ;  }  main()  {  int i, M, T[100] ;  for(i=0; i<10; i++)  scanf("%d", &T[i]) ;  M=T[0] ;  for(i=1; i<10; i++)  M=max(M , T[i]); // call function  printf("The maximum of the array is: %d", M) ;  } |

1. **C procedures** 
   1. **Definition**

**voidaction\_name**(list-params)

{

list-declarations (optional)

list\_instructions

}

* 1. **Call**

action\_name(expression-list)

* **You can't retrieve the result of a procedure - there isn't one.**
  1. **Example**

Let's take the same example:

***Definition of the procedure***

void max(int a , int b)

{

int m;

i f (a>b)

m=a ;

else

m=b ;

printf(''%d'', m) ;

}

***Call***

max(3 ,45);

To use this procedure to display the maximum of an array, we need to pass parameters by address; the program is then written as follows:

|  |
| --- |
| #include<stdio.h>  void max ( int a , int b , int \*c)  {  int m;  if (a>b)  m=a ;  else  m=b ;  \*c=m;  }  main()  {  int i, M, T[10] ;  for(i=0; i<10; i++)  scanf("%d", &T[i]) ;  M=T[0] ;  for(i=1; i<10; i++)  max(M , T[i], &M); // call procedure  printf("The maximum of the array is: %d", M) ;  } |

When passing a parameter by value, there's nothing new, as shown in the procedure definition and call example; whereas when passing a parameter by address, the formal parameter(s) you wish to return the result(s) to are preceded after their type(s) by **(\*)**, and the effective parameter(s) that receive(s) this/these result(s) are preceded by **(&).**

**Example**:

|  |
| --- |
| #include<stdio.h>  void fact(int y ,int **\*z**)  {  int t=1, j;  for(j=1;j<=y;j++)  t=t\*j;  \*z=t;  }  main()  {  int b;  fact(4,**&b**);  printf("%d", b);  } |

* 1. **Exercises**

Write the following procedures:

* Printing of up to 3 integers in parameters
* Print the first 100 terms of the following sequence:u0 = 32, un = 3∗ un-1 + 19
* Print the first k terms of the sequence, with k passed as a parameter.

1. **Passing parameters by address in functions**

To return more than one result in functions, one of the results is returned by return(); the others are returned using address passing, just like procedures.

**Example**: calculate the sum and product of a sequence from 1 to a, i.e. (1+2+ ....+a) and (1\*2\* ....\*a).

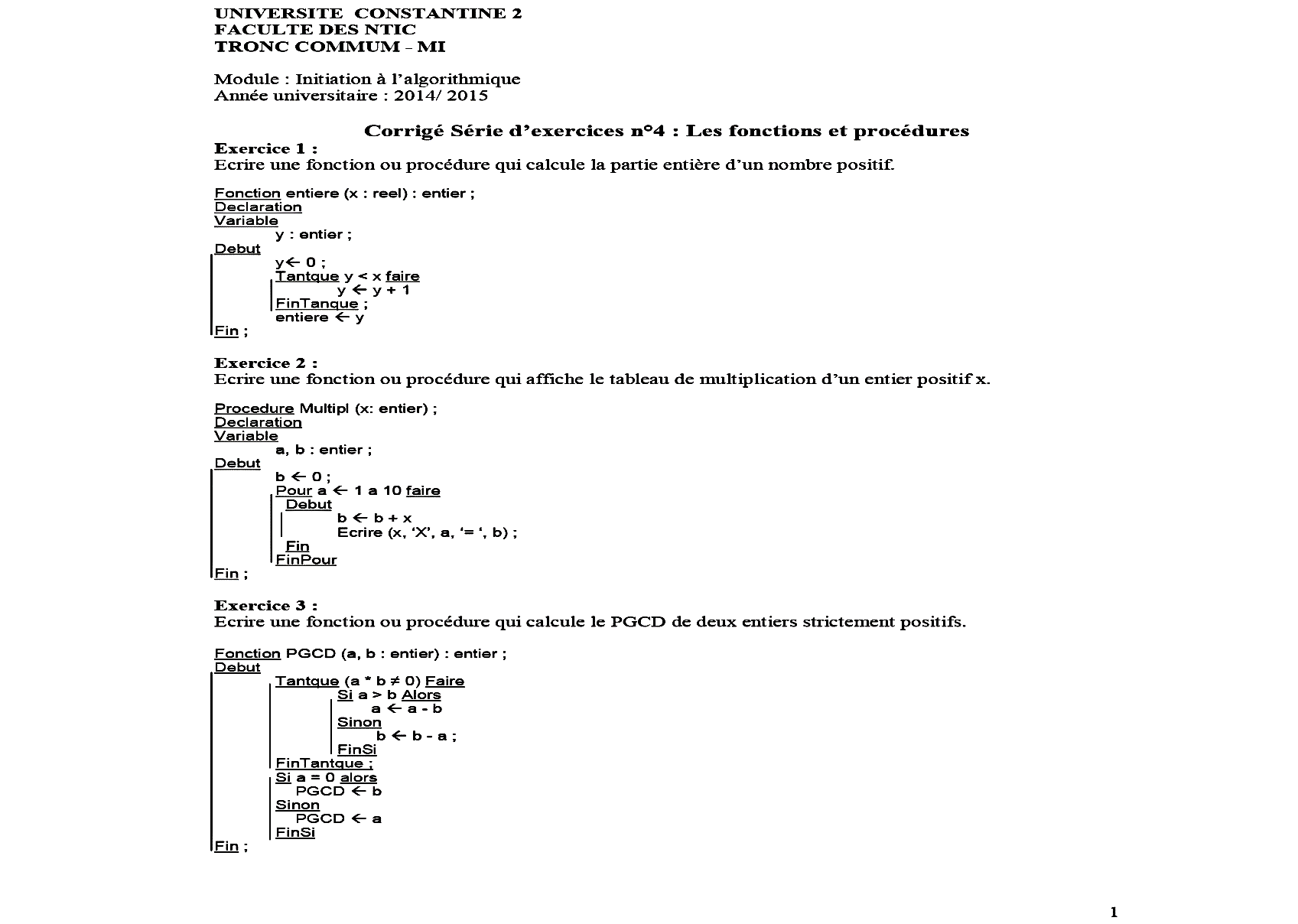
|  |
| --- |
| #include<stdio.h>  int fact(int x, int **\*y**)  { int r,r2,i;  r=1 ;  r2=0 ;  for(i=1;i<=x;i++)  {  r=r\*i;  r2=r2+i;  }  **\*y=r2**;  return(r); // places itself last  }  main()  {  int a=4,b;  a=fact(a , **&b**); printf( "%d%d" , a , b ); // after execution displays two results: 24 and 10  } |

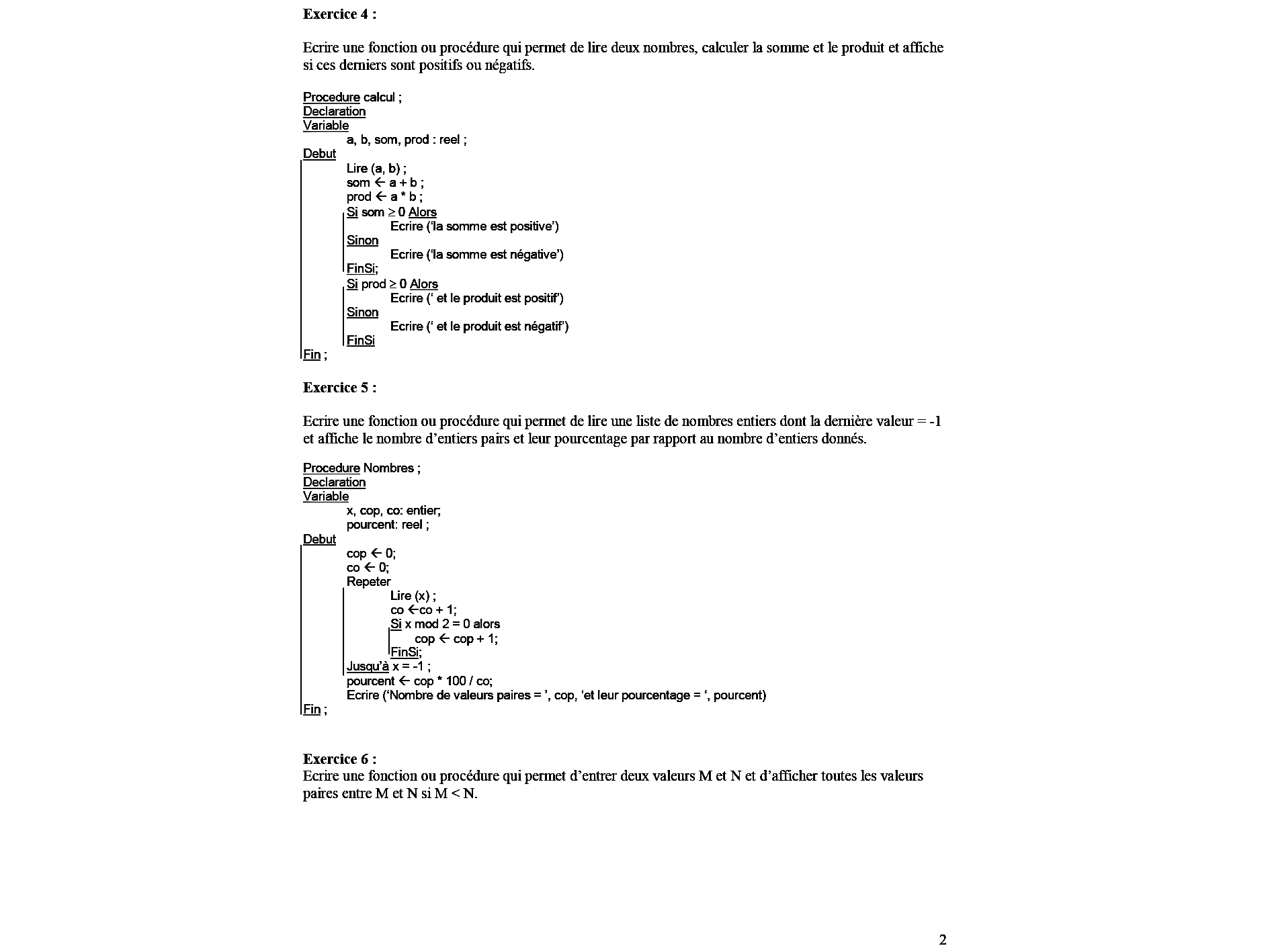
**RQs**:

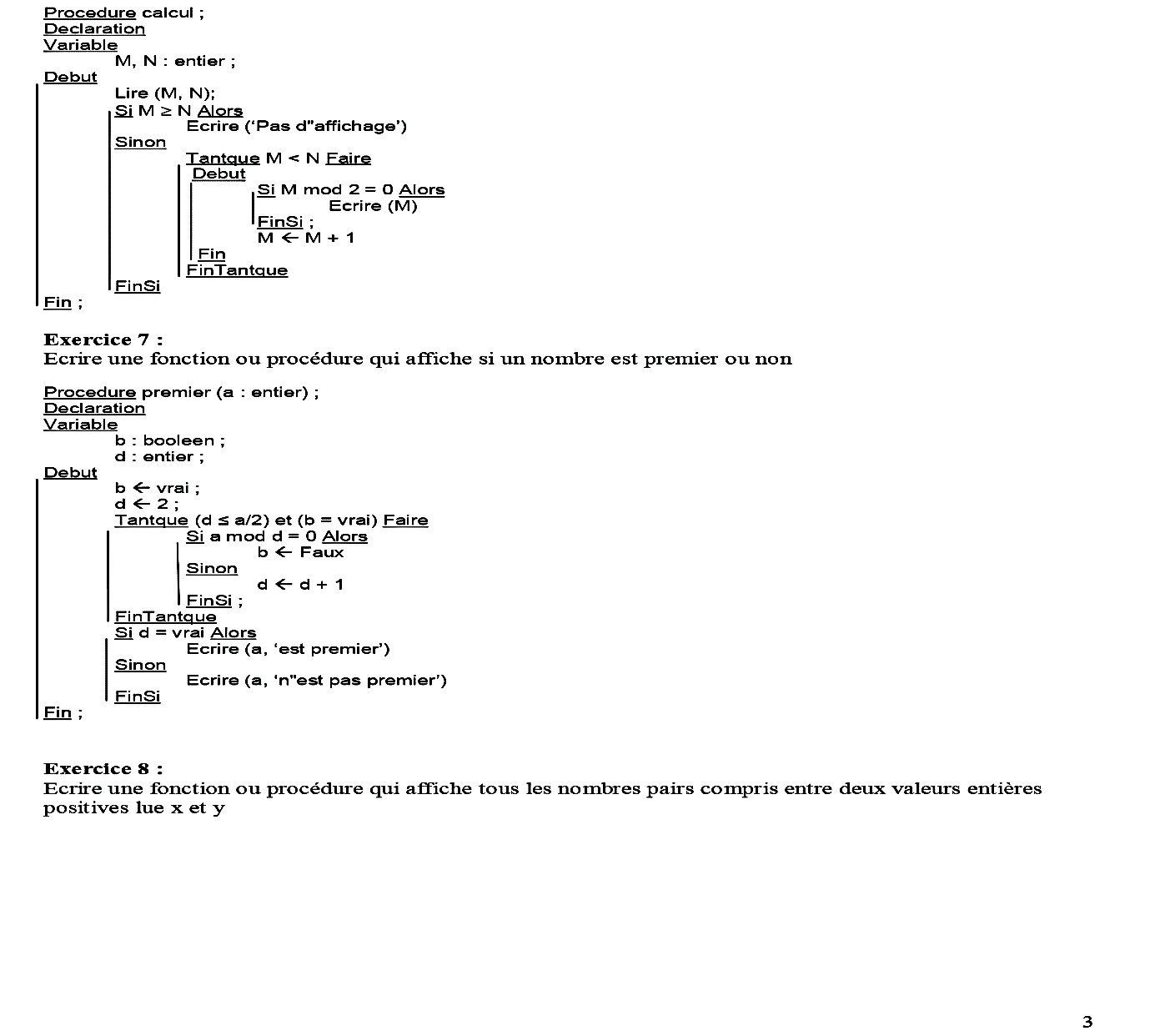
* Declare a parameter as "value" if you don't want the call to modify the arguments (effective parameters).
* Declare a parameter as a "reference" if you want the call to modify the arguments
* Of course, you can mix parameters by value and by reference.

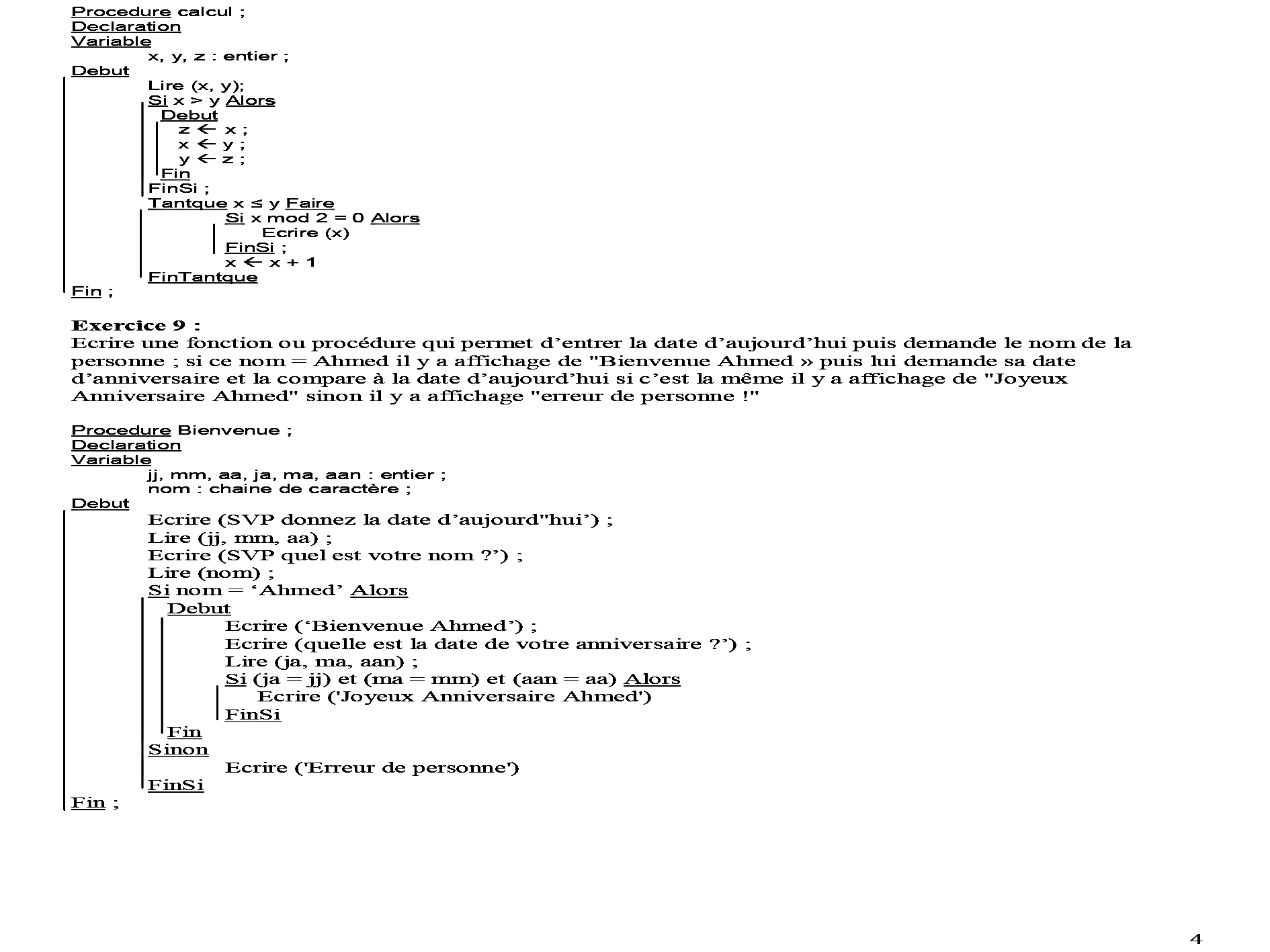
**Corrected exercises : Functions and procedures**

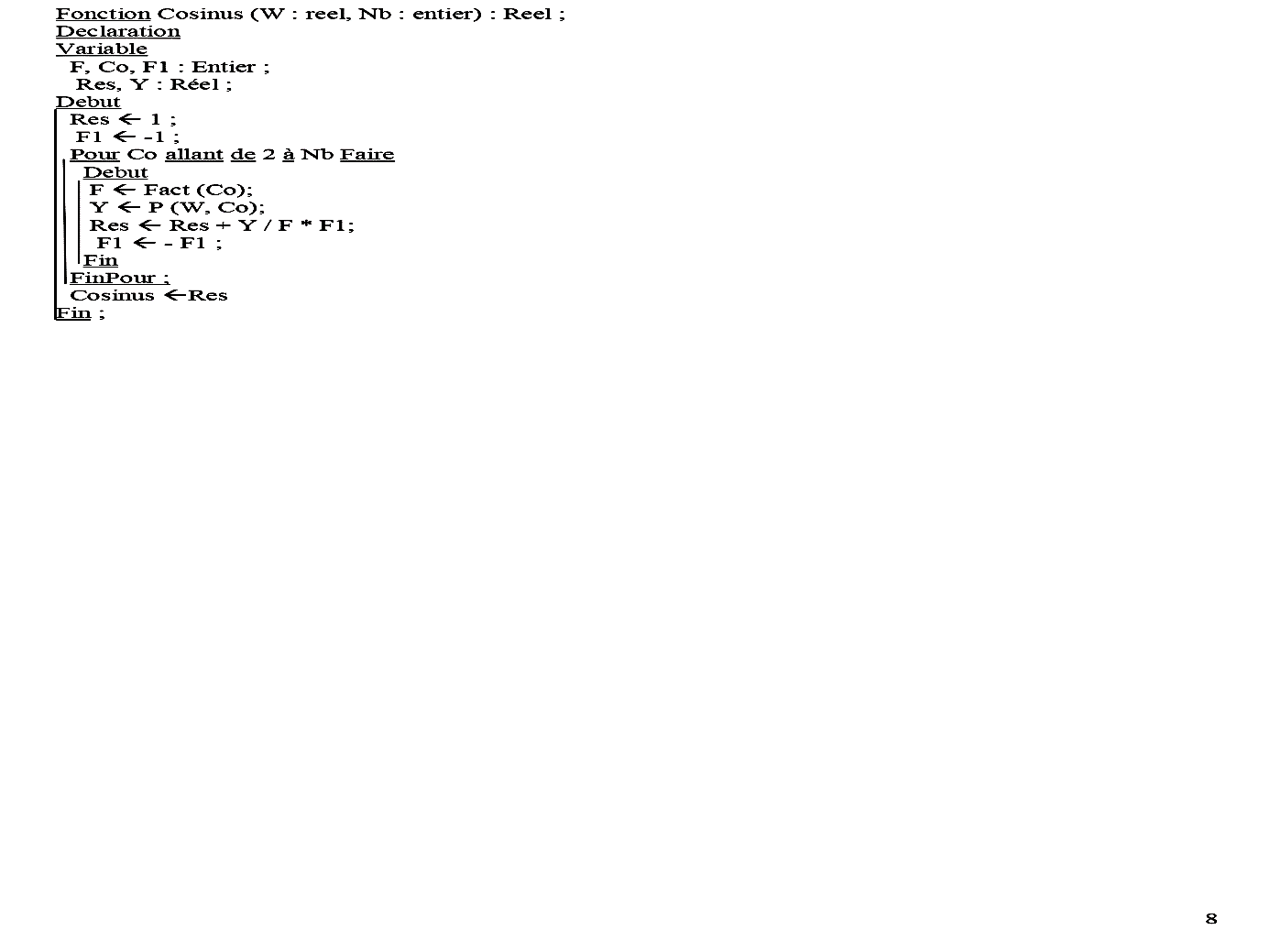
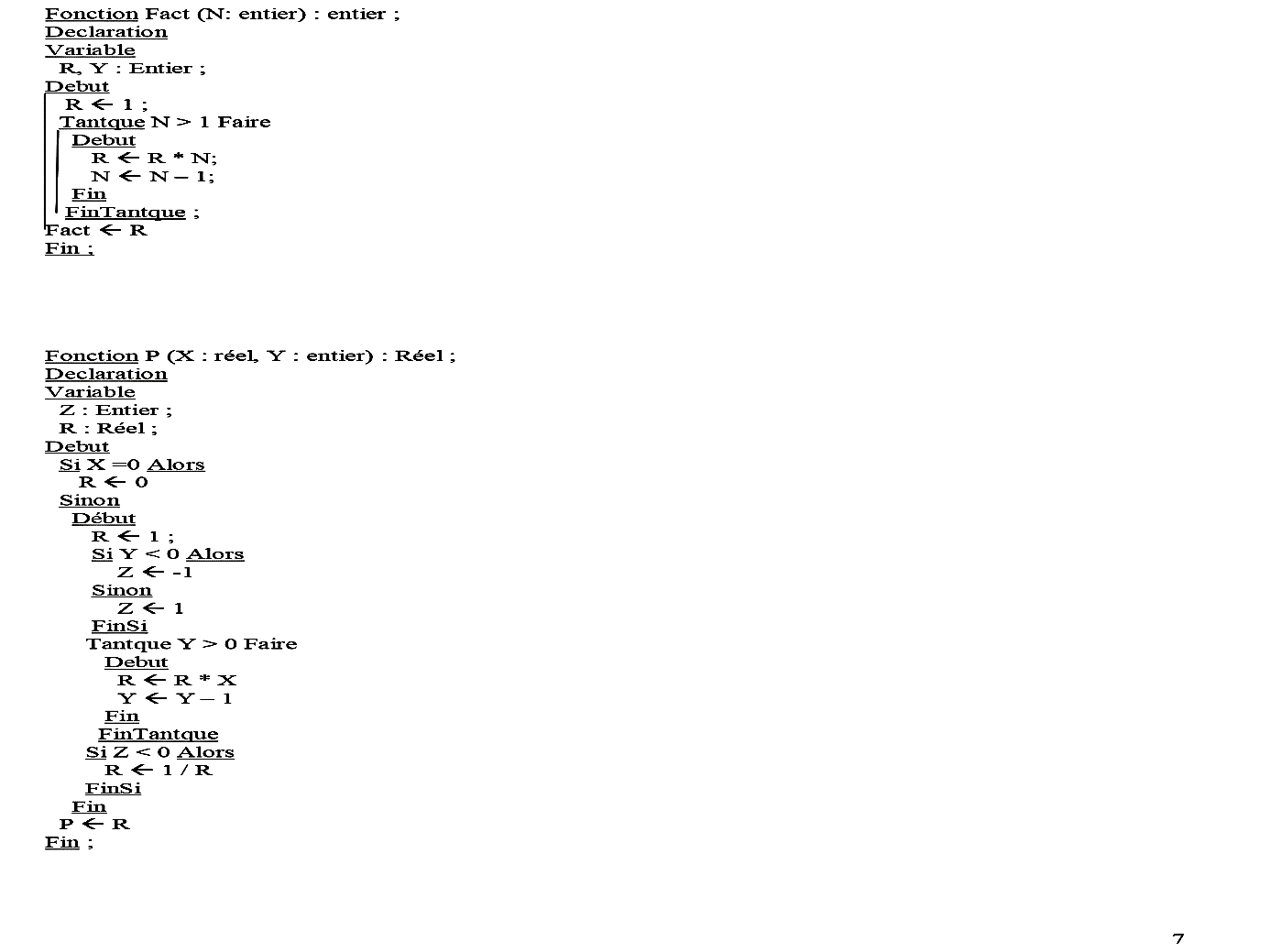
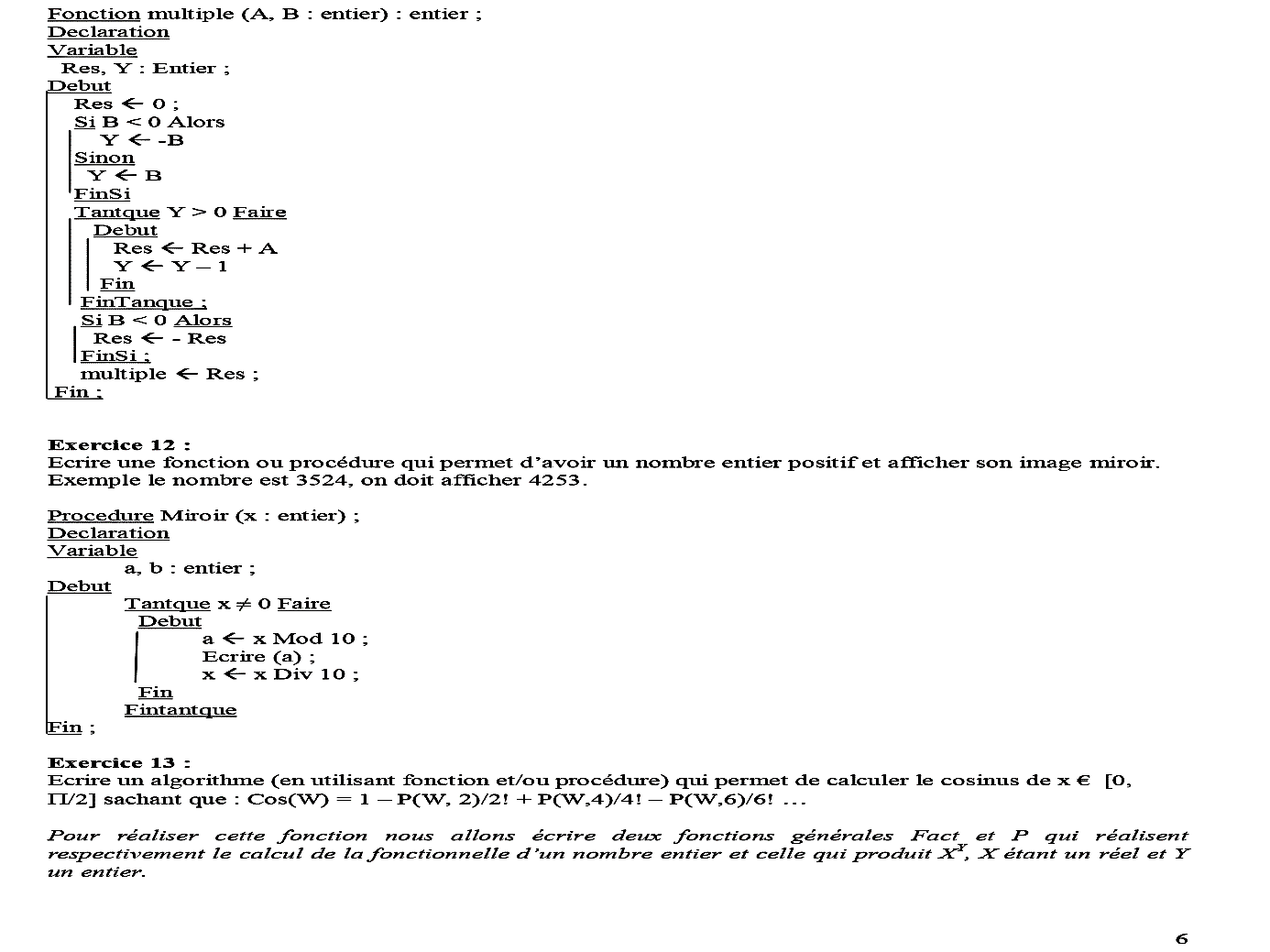
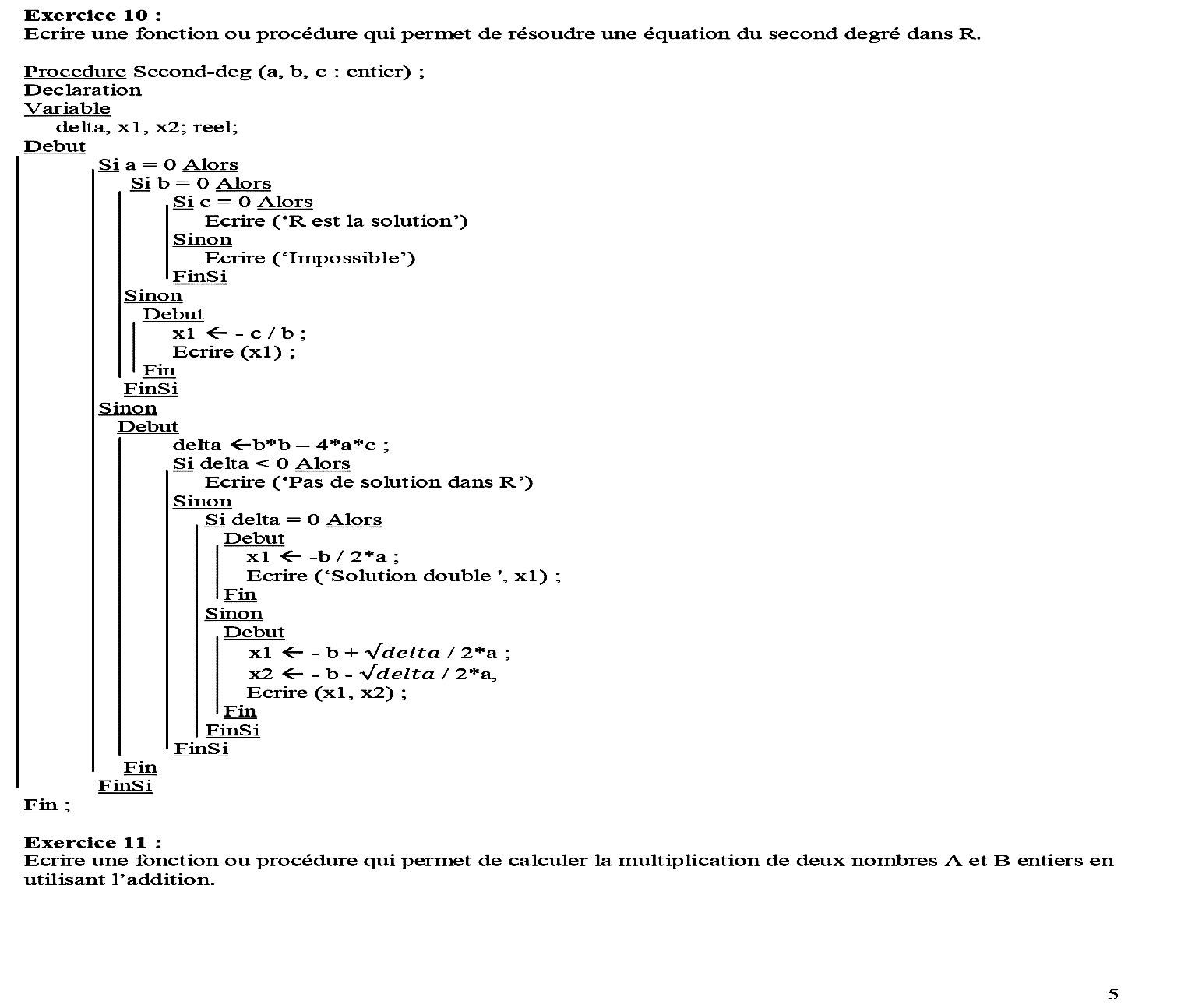
*(universiteconstantine 2 tronc commum - mid 2014/2015)*











# **Home exercises**

**Exercise 1**

Let N be a positive real number.

* Write a function or procedure that determines the integer part of "N".
* Write a function or procedure that determines the fractional part of "N".
* Write a function or procedure that determines the integer and fractional parts of "N".

**Exercise 2**

Let T be an array of real "M" (M≤35).

* Write procedures or functions that allow :
* Read T values.
* Display T values.
* Determine the index of the maximum value in table T.
* Determine the index of the maximum value in a part of the T array.
* Exchange the values of two cells in table T.
* Sort table T in ascending order.
* Using the necessary functions and procedures, write **an algorithm** to read the grades of a group of "N" students (N≤35), sort them in ascending order and then display them.

**Exercise 3**

* Write a function or procedure that reads two numbers, calculates their sum and product and displays whether they are positive, zero or negative.

**Exercise 4**

* Write a function or procedure to solve a second-degree equation in R.

**Exercise 5**

* Write a function to check whether an array is sorted in ascending order or not.

**Exercise 6**

Let A be a real matrix with N rows and M columns, N≤100, M ≤100, and N is even.

* Write a function or procedure that determines the minimum of line "k".
* Write a function or procedure that calculates the sum of the elements in line "k".
* Using the two subroutines above, write **an algorithm** that calculates, for any pair of 2 consecutive rows (1 and 2; 3 and 4; 5 and 6; ... N-1 and N), the quotient of the sum of the elements of the odd-numbered row and the minimum of the even-numbered row.

**Exercise 7**

Using the functions (factorial and power) from Exercise 1, write an algorithm to calculate the values of the following expressions:

**Exercise 8**

* Write a function or procedure that displays the multiplication table of a positive integer x.

**Exercise 9**

Write a function or procedure that calculates the PGCD of two strictly positive integers.