Chapter 1. Subprograms: Functions and Procedures

Dr. Abderrahmane Kefali

Senior Lecturer Class A,

Department of Computer Science,

University of May 8, 1945 - Guelma

kefali.abderrahmane@univ-guelma.dz

This document is prepared for printing two pages per sheet

1) Introduction

When dealing with a complex problem, it is more desirable to decompose it into a set of less complex sub-problems and establish a solution, in the form of a set of instructions or modules, for each of these sub-problems. The blocks or modules that solve sub-problems are called ***subprograms***. The algorithm solving the initial problem is then formed by the set of subprograms solving the sub-problems. This is the principle of ***structured*** and ***modular programming***.

In this chapter, we will present the advantages of modularity, discover how we can define and use our own subprograms in algorithmics and in the C language.

2) Definitions

2.1) Subprogram

2.1.1) What is a Subprogram

A subprogram is a functional unit consisting of data and a block of instructions. It is named and optionally parameterized, designed to perform a specific task. A task is a well-defined action, more or less complex, that occurs at a given moment and involves one or more objects. The subprogram is declared before the start of the algorithm and will only be executed after being called within the algorithm's body. The collection of interconnected subprograms must be capable of solving the overall problem.

In algorithmics, there are two types of subprograms: ***procedures*** and ***functions***.

2.2) Function

A function is s a subprogram, that can take zero, one, or multiple parameters and always returns a unique scalar result. The function is invoked by mentioning its name whenever we want to use it.

The main purpose of a function is to return a result after receiving parameters as input and executing multiple associated instructions.

2.3) Procedure

A procedure is a subprogram that takes zero, one, or more parameters and does not return a result.

However, it is possible to pass data back to the calling program through parameters passed by reference (see section 6.2).

3) Functions: Declaration and Invocation

3.1) Functions in Algorithmics

3.1.1) Declaration and Definition

The declaration of a function involves announcing that a particular identifier corresponds to a function, which returns a certain type and takes certain parameters. The definition of a function is a declaration where, in addition, the code of the function is provided.

a) Function Definition

A function is defined within the declaration section of an algorithm. The structure of a function consists of a header, a declaration part, and a body of the function (processing part). The syntax for defining a function is as follows:

```
Function <function_name> (<par_1>:<type_1>,...<par_n>:<type_n>):<return_type>;
<declaration_of_local_objects>
Begin
<block_of_instructions>
End;
```

This definition comprises three parts:

- The first line serves as the header or signature of the function. In this header, **`<function_name>`** is the name of the function, and **`<return_type>`** denotes the type of the value returned by the function. The header also indicates, within parentheses, the list of argument declarations separated by commas. Thus, **`<par_i>`** is an identifier representing the name of the ith parameter, and **`<type_i>`** is its type.
- Immediately after the header is the declaration part, where all objects (constants, variables, custom types) local to the function are described. Any object defined or declared at this level is specific to the function and will not be recognized outside the function's scope.
- The remainder of the definition corresponds to the body of the function. This is a block containing instructions that will be executed when the function is called.

Remarks:

- The parameters of the function are called formal parameters because the values they represent are undetermined until the function is called (executed).
- A function can have no arguments (formal parameters).
- If multiple parameters are of the same type, it is possible to factor out their common type.
- Formal parameters can be of any simple, structured, or custom type. However, the return type of the function must be a scalar type.
- A function can include in its declaration part other subprograms that, in turn, may contain additional subprograms, creating a hierarchical structure.

Examples:

The following headers are correct:

```
Function sum(x: real, y: real): real;
Function nbPositives(T: Array[10] of integer): integer;
Function mid_Point(P1, P2: Point): Point;
Function read_Character: Character;
```

b) Function Return

A function must return a value. To achieve this, the body of the function must conclude with a return statement that delivers the result calculated by the function.

In algorithmics, the return is accomplished using an assignment statement that assigns the result to the function's name. It takes the form:

```
<function_name> ← <result>;
```

Where **`<result>`** can be a direct value, a variable, or an expression. It must be of the same type as the return type specified in the function's header.

Example:

The following function **`average`** calculates the average of three integer numbers passed as parameters:

```
Function average(a,b,c:integer):real;
Var avg:real;
Begin
avg ← (a+b+c)/3;
average ← avg;
End;
```

3.1.2) Function Invocation

Once a function is declared, it can be invoked (or called) from the main algorithm or another subprogram. In fact, the block of instructions comprising the body of the function will only execute when the function is called.

a) How to Call a Function

The call is made by specifying the name of the called function and optionally providing a list of expressions in parentheses corresponding to the values of the various formal parameters of the function. The parameters used during the function call are called ***actual parameters***. Since the function returns a single value, a variable to which the returned result is assigned (output variable) is also specified. The syntax for calling a function is as follows:

```
<variable> ← <function_name>(<par1>,...,<parn>);
```

Examples:

The following calls are correct:

```
F ← Fact(5);
s ← sum(x+2,y-4);
c ← read_Character;
```

Remarks:

- The call of a function is an expression, and its value is the result returned by the function. Therefore, the function can be called within an expression, assignment, writing statement, condition, and so forth.
- In the case of a function without parameters, the list of expressions between parentheses must be empty.
- Formal parameters and actual parameters must match in number, type, and order. Their names can differ.
- Actual parameters can be expressions, constants, or variables.
- The types of the output variable and the result returned by the function must be compatible.

Example:

The following instructions contain function calls:

```
Write("The sum is ", sum(5, 3));
R ← (Fact(6) + 10) / 2;
If read_Character = 'M' then Write("The person is male")
Else Write("The person is female");
```

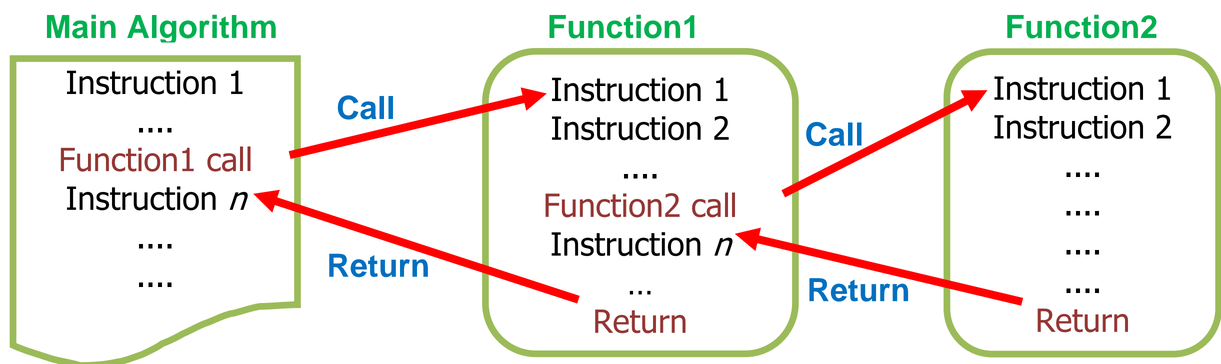
b) What happens during a function call

The process of calling a function proceeds as follows:

- Save the address of the instruction following the call (the return address).
- Match each formal parameter to the corresponding actual parameter.
- Execute the instructions of the called function.

- Upon completion of the function, return to the calling algorithm and resume the execution of its instructions starting from the return address.

The following figure illustrates this process.



3.1.3) Formal Parameters and Actual Parameters

Parameters play a crucial role in functions, facilitating communication with the main algorithm and other subprograms. It is through parameters that a function can be executed multiple times with different values.

However, a distinction is made between formal parameters and actual parameters. The parameters described in the function's header during declaration are called formal parameters. They serve to represent the structure and functioning of the function. On the other hand, the parameters specified when calling a function are called actual parameters or arguments. These are the values that will be used during the execution of the function.

3.2) Functions in the C language

3.2.1) Declaration and definition

a) Definition

A function in the C language consists of two parts. The first is the header, which describes the function's name, return type, and the list of its formal parameters along with their types. The second part is the body of the function, which contains the instructions to be executed by the function and potentially the declaration of local objects (variables, types, etc.).

Thus, the definition of a function in the C language follows the syntax:

```
<return_type> <function_name>(<type1> <par1>,...,<typen> <parn>)
{
   <declaration_of_local_objects>
   <block_of_instructions>
}
```

Remarks:

- There is no semicolon after the closing parenthesis "**)**" of the header.

- Formal parameters can be of any type, while the function's return type must be a scalar type.
- It is not allowed to "factorize" a common type for multiple arguments. Each argument must be preceded by its type.
  For example, **`float sum(float a, b)`** is not allowed. The type of each argument must be specified, such as : **`float sum(float a, float b)`**.
- When a function has no parameters, you can either leave the parentheses empty (with parentheses nonetheless) or explicitly indicate that it has no parameters by using **`void`** keyword.
- If the function's return type is not indicated, the compiler assumes it is an integer result function (**`int`** type).
- Unlike in algorithmics, in C, you cannot define a function inside another function; all functions are at the same level.

Examples:

The following headers are correct:

```
float sum(float x,float y)
int nbPositives(int T[10])
Point mid_Point(Point P1,Point P2)
char read_Character()
```

b) The `return` Statement

The instruction used to transfer control back to the calling function in the C language is the **`return`** ***statement***. The body of the function must contain at least one **`return`** statement, and its syntax is as follows:

```
return <expression>;
```

The value of **`<expression>`** is the value returned by the function. Its type must match the type specified in the function's header.

Example:

The following **`average`** function calculates the average of three integers passed as parameters:

```
float average(int a,int b,int c){
  float avg;
  avg = (float)(a+b+c)/3;
  return avg;
}
```

Remarks:

- Multiple **`return`** statements can appear in a function. The return to the calling program will be triggered by the first **`return`** encountered during execution.

- The **`return`** statement is like any other instruction, so it is possible to use as many as desired within the body of a function.

Example:

The following function takes 2 integer parameters and returns the maximum of them.

```
int max(int a, int b) {
    if (a > b) return a;
    else return b;
}
```

3.2.2) Function invocation in C

Invoking a function in the C language is done similarly to algorithmic languages. The syntax for the invocation is as follows:

```
<variable> = <function_name>(<param1>,...,<paramn>);
```

Examples:

The following invocations are correct:

```
int result = Fact(5);
float M = average(x + 2, y - 4, 7);
char c = read_Character();
```

Remark:

In the case of calling a function without parameters, the parameter list inside the parentheses must be empty; it is not possible to use the **`void`** keyword.

4) Procedures: Declaration and Invocation

4.1) Procedures in algorithmics

4.1.1) Declaration et definition

Similar to a function, a procedure must be declared before being used, and this is done in the algorithm's header.

Syntaxically, the definition of a procedure is as follows:

```
Procedure <procedure_name> (<par1>:<type1>,...<parn>:<typen>);
<declaration_of_local_objects>
Begin
<block_of_instructions>
End;
```

Where:

- **`<procedure_name>`** is the name of the procedure.
- **`<par_i>`** is the name of the i-th formal parameter, and **`<type_i>`** is its type.

- **<declaration_of_local_objects>**: Objects declared here are specific to the procedure and are only recognized within the procedure.
- **<block_of_instructions>**: This is the sequence of actions constituting the body of the procedure.

Remarks:

- A procedure does not include a return statement.
- A procedure may have no formal parameters.
- If multiple parameters are of the same type, it is possible to factorize their type.
- Formal parameters can be of any simple, structured, or custom type.
- A procedure can include in its declaration section other subprograms, which in turn can contain other subprograms, and so on.

Examples:

- The following procedure, **sum**, allows the user to input two real numbers, calculates their sum, and displays the result:

```
Procedure sum;
Var a, b, s: real;
Begin
Write("Enter 2 real numbers:");
Read(a, b);
s ← a + b;
Write("The sum is ", s);
End;
```

- The next procedure, **average**, calculates and displays the average of three integers passed as arguments:

```
Procedure average(a, b, c: integer);
Var avg: real;
Begin
avg ← (a + b + c) / 3;
Write("The average is ", avg);
End;
```

4.1.2) Procedure Invocation

To execute a procedure, it is sufficient to call it by writing its name followed by the actual parameters, following the syntax below:

<procedure_name>(<par$_1$>,...,<par$_n$>);

Where **<par$_i$>** is the ith actual parameter.

Examples:

The following calls are correct:

```
sum;
average(x, y*2, 5);
```

Remarks:

- The invocation of a procedure triggers the same actions as when calling a function. These actions are described in section 3.1.2.b.
- The invocation of a procedure is an independent instruction, and it cannot be used within another instruction or expression.
  For example, the call: **`m ← average(5,3,9)`** is incorrect.
- Formal and actual parameters in procedures follow the same rules as those in functions.

4.2) Procedures in C language

A procedure in C language is a type of function that does not return any value.

4.2.1) Declaration and definition

In the C language, the declaration of a procedure translates to the declaration of a function with a return type of **`void`**. It follows the syntax below:

```
void <procedure_name>(<type1> <par1>, ..., <typen> <parn>)
{
    <declaration_of_local_objects>
    <block_of_instructions>
}
```

Remarks:

- The **`return`** statement is not mandatory. If present without parameters in a procedure, it serves to terminate its execution.
- All remarks regarding the header, formal parameters, local variables, for functions in the C language, remain valid for procedures as well.

Examples:

- The following procedure, **`sum`**, reads two real numbers, calculates their sum, and displays the result:

```
void sum(){
  float a, b, s;
  printf("Enter 2 real numbers:");
  scanf("%f%f", &a, &b);
  s = a + b;
  printf("The sum is %f", s);
}
```

- The following procedure, **average**, calculates and displays the average of three integers passed as arguments:

```
void average(int a, int b, int c){
  float avg;
  avg = (a + b + c) / 3.0;
  printf("The average is %f", avg);
}
```

4.2.2) Procedure invocation

The syntax for calling a procedure in the C language is as follows:

```
<procedure_name>(<par_1>,...,<par_n>);
```

Examples:

The following calls are correct:

```
sum();
average(x, y * 2, 5);
```

Remarks:

- Formal and actual parameters in procedures follow the same rules as those in functions.
- In the case of calling a procedure without parameters, the parameter list within parentheses must be empty; it is not possible to use the **void** keyword as an actual parameter.

5) Local and global variables

In algorithmics, not all variables have the same lifespan. There are permanent variables that occupy a memory location that remains the same throughout the program's execution, and temporary variables that occupy a memory location dynamically during program execution.

In fact, the lifespan of variables is intricately related to their ***scope***, meaning the portion of the program in which they are defined. This distinction results in the differentiation between two types of variables: global variables and local variables.

5.1) Global Variables

A ***global variable*** is defined as any variable declared within the main algorithm and outside of any subprogram. Global variables are recognized by the compiler throughout the code segment following their declaration. They are modifiable and accessible by all subprograms without the necessity of passing them as parameters. Consequently, their scope (or validity space) is global. It is noteworthy that global variables are consistently permanent.

5.2) Local Variables

Variables declared within a subprogram are called ***local variables***. By default, local variables are temporary. Their lifespan is limited to the execution of the subprogram, and their scope is therefore confined to that subprogram. When a subprogram is called, it places its local variables on the stack. At the exit of the subprogram, local variables are removed and thus lost.

Local variables are known and accessible only within the subprogram where they are declared. They cannot be used by the main algorithm or any other subprogram. To modify or use them in another subprogram, it is necessary to pass them as parameters.

Remarks :

- Local variables in a function have no specific connection with global variables of the same name or local variables in another function with the same name.
- When the name of a local variable is identical to the name of a global variable, the global variable is locally shadowed, meaning it becomes inaccessible in that subprogram.
- Global variables should be used with caution as they create invisible links between subprograms.
- In the C language, variables declared inside the **`main`** function are local to the **`main`** function and are not global variables. Global variables in C must be declared outside of any function.

6) Parameter passing

When calling a subprogram, formal parameters are substituted with actual parameters. This operation is called "parameter passing". There are two modes of parameter passing: ***pass by value*** and ***pass by reference (or address)***. These two modes of parameter passing are related to whether the subprogram has the right to modify a value passed to it as a parameter or whether it can only use it in read-only mode.

6.1) Pass by value

In algorithmics and the C programming language, by default, parameters are passed by value. In this mode of parameter passing, only the value of the parameter is considered, not its memory address.

In fact, the formal parameters of a subprogram are treated in the same way as local variables. Thus, at the time of a subprogram invocation, the actual parameters are copied into the memory locations of the corresponding formal parameters. The subprogram then operates only on the local copy of the actual parameters. This copy disappears upon returning to the calling program. This implies, in particular,

that if the subprogram modifies the value of one of its parameters, only the copy will be modified; the variable in the calling program will not be affected.

Example:

This algorithm is supposed to reset a number passed as a parameter (by value).

```
Algorithm example;
Var x: integer;
Procedure reset(a: integer);
Begin
a ← 0;
Write("a=", a);
End;
Begin
x ← 10;
reset(x);
Write("x=", x);
End.
```

This algorithm will display:

a=0

x=10

This indicates that the modification was made only to the local variable **a**, which took a copy of the global variable **x** without affecting the latter.

6.2) Pass by reference (address)

Passing by reference involves passing not the values of the parameters but the variables themselves (their addresses). Thus, each formal parameter contains the address of the corresponding actual parameter and not its value. This mode of parameter passing implies that the called subprogram will use this variable as if it were locally declared. Therefore, any modification of the parameter transmitted in the called subprogram automatically results in the same modification to the variable passed as a parameter in the main algorithm.

6.2.1) In algorithmics

In algorithmics, to pass a formal parameter by address, you simply declare it in the header of the subprogram using the **Var** keyword.

The syntax of the call remains unchanged.

Example:

The following algorithm contains a procedure that swaps two integers.

This procedure must modify the two values passed to it as parameters. Therefore, these values must be passed by reference:

```
Algorithm example;
Var x, y: integer;
Procedure swap(Var a, b: integer);
Var c: integer;
Begin
c ← a;
a ← b;
b ← c;
End;
Begin
x ← 5;
y ← 3;
swap(x, y);
Write("After swap x=", x, " and y=", y);
End.
```

Remarks:

- Passing by variable is very useful, especially with procedures, when we want our procedure to indirectly return one or more results. In this case, the procedure returns the expected results in parameters passed by address.
- Passing arrays as parameters is always done by reference.
- The same subprogram can use both modes of parameter passing. In this case, the two modes of passing must be separated by a semicolon "`;`".

6.2.2) In C language

In C language, passing by address differs from passing by value in declaration, usage, and invocation.

a) In declaration

In C language, to indicate that a formal parameter is passed by reference, you simply add an asterisk "**`*`**" after the type in the subprogram's declaration.

For example, to pass an integer parameter **`a`** by reference, the formal parameter is declared as: **`int* a;`**

b) In use

Inside the subprogram, parameters passed by reference are manipulated using the ***indirection operator***, denoted by "**`*`**". This operator, when applied to an address, allows access to the content located at that address.

For example, to use the parameter **a** passed by reference inside a subroutine, you write ***a**.

c) In invocation

Invoking a subprogram using parameters passed by variable must be done by passing the addresses of the variables to be modified using the address-of operator **"&"**.

Example:

The C program that performs the permutation of two values using a procedure with parameters passed by variable is as follows:

```
void swap(int* a,int* b){
  int c;
  c = *a;
  *a = *b;
  *b = c;
}
main(){
  int x,y;
  x = 5;
  y = 3;
  swap(&x,&y);
  printf("After swap x=%d and y=%d",x,y);
  return 0;
}
```

7) Recursion

Recursion is a simple and elegant way to solve certain algorithmic problems, especially in mathematics. However, it requires an understanding of how this principle works. Recursion is used with problems that are inherently recursive.

7.1) Definitions

7.1.1) Recursive Problem

A problem is considered recursive when its resolution depends on the solution to itself at a lower order.

Examples:

The following problems are recursive:

- Calculating factorial: $n! = n \times (n-1)!$
- Calculating power: $A^n = A \times A^{n-1}$
- Calculating the greatest common divisor (GCD)
- Fibonacci sequence.

7.1.2) Recursive subprogram

The simplest definition of a recursive subprogram is as follows: *it is a subprogram that calls itself during its execution*. If you call a subprogram within its own body (content), then this subprogram is recursive.

The call of a subprogram inside itself is called a ***recursive call***.

Remark :

A subprogram that is not recursive is called ***iterative***.

7.2) First example

The usual and most well-known example of recursion is the calculation of a factorial. The factorial of a positive integer **n** is denoted as **n!** and is equal to the product of integers from 1 to **n**, inclusive:

n! = n × (n-1) × (n-2) × … × 2 × 1.

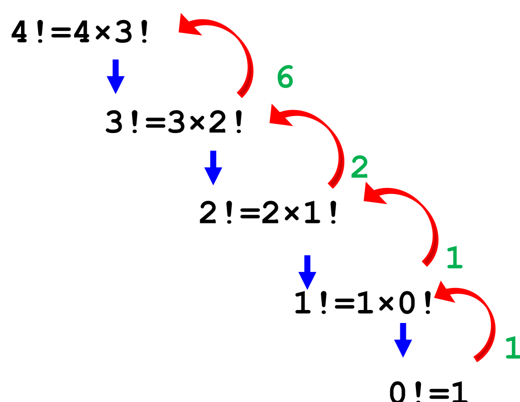
An iterative definition of the function **Fact** (which takes a positive integer **n** as input and returns **n!**) is as follows:

```
Function Fact(n:integer):integer;
Var r,i:integer;
Begin
r ← 1;
For i ← 1 To n Do
 r ← r * i;
Fact ← r;
End;
```

In fact, from the definition of the factorial, we can extract the following recurrence relation: **n! = n × (n-1)!**

So, **n!** is calculated in terms of **(n-1)!**. Similarly, **(n-1)!** is calculated in terms of **(n-2)!**, and so on until reaching **0!**. By definition, **0!** is equal to 1, and the value 0 determines the end of the recursion.

For example, to calculate **4!**:



However, a recursive definition of the **Fact** function (based on the previously mentioned recurrence relation) can be provided as follows:

```
Function Fact(n:integer):integer;
Begin
If n=0 then Fact ← 1
Else Fact ← n * Fact(n-1);
End;
```

Here, it can be observed that the final return is actually a ***recursive call***, subtracting 1 with each call until **n** equals 0, which, as described earlier, is our ***exit condition*** marking the end of recursion.

Thus, the **Fact** function is invoked multiple times, and with each call (to calculate **Fact(n)**), the ongoing computation is paused to calculate the result of **Fact(n-1)** before returning to the current computation.

7.3) How to Write a Recursive Subprogram?

To write a recursive subprogram:

- First, handle the base case, i.e., the one that does not require a recursive call. For the **Fact** function, for example, this is the case where **n = 0** (as **Fact(0)** is directly known to be 1).
- Next, address the recursive subproblems by invoking the recursive function for each sub-problem that needs resolution.

7.4) Structure of a Recursive Subprogram

A recursive subprogram is defined similarly to an iterative subprogram; the key distinction lies in the inclusion of a termination condition and at least one self-invocation within its definition. The structure of a recursive function is as follows:

```
Function <function_name> (<par1>:<type1>,...,<parn>:<typen>): <return_type>;
<declaration_of_local_objects>
Begin
......
If <condition> Then      // Termination condition, base case, ...
   <function_name> ← <expression>
Else Begin
     <function_name> ← <function_name>(...);   // Recursive call
     ......
     <function_name> ← <function_name>(...);   // Recursive call
     ......
     End;
End;
```

Remarks :

- Every recursive subprogram must account for multiple cases, with at least one not involving a recursive call (base case), preventing infinite recursion.
- Each recursive call ideally «approaches» a base case, ensuring program termination.
- Any recursive subprogram can be replaced by its iterative equivalent, but the reverse is not always true.
- In a recursive solution, the problem is tackled in reverse. Take the factorial as an example; the approach starts from the number and works backward to 0, the base case, enabling factorial calculation.
- Recursive programming, for certain problems, is highly efficient for the programmer; it enables the correct implementation with minimal instructions.

7.5) Call Stack

The execution of a recursive subprogram doesn't conclude until all subproblems are resolved. Throughout this process, the execution context (parameters and local variables of that call) is stored in a stack, called the ***call stack.***

Take the example of the **`Fact`** function for **`n`** equal to 4. Recursive calls inside the function stack up: **`Fact(4)`** calls **`Fact(3)`**, waits for **`Fact(3)`** to finish, then finishes itself. Of course, **`Fact(3)`** will have called **`Fact(2)`**, which itself calls **`Fact(1)`**, which calls **`Fact(0)`**, which calls no one.

Note that recursive subprogram calls are ***stacked*** in Last In First Out (LIFO) order. Each call is therefore stacked one after the other.

When reaching the end, the calls are ***popped*** off the stack, enabling the step-by-step calculation of the value of **`Fact(4)`**.

The execution of subprogram calls occurs in the reverse order as they are removed from the stack.

Hence, a recursive subprogram involves two phases: the ***stacking*** phase (descent) and the ***popping*** phase (ascent).

7.6) Other Examples of Recursive Problems: Tower of Hanoi

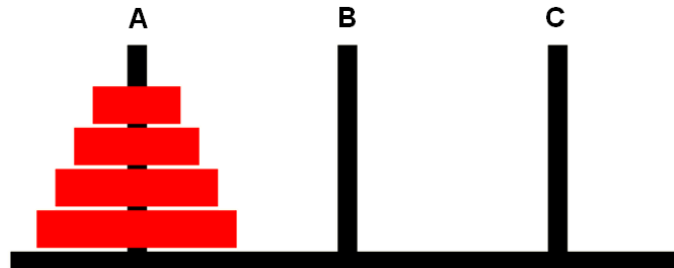
7.6.1) Description

The "Tower of Hanoi" is a puzzle conceived by the French mathematician Edouard Lucas (1842-1891). The objective is to move **`n`** disks of different diameters stacked in descending order on a "source" tower to a "destination" tower, passing through an "intermediate" tower, using the minimum number of moves.

There are two rules to follow:

1. Only one disk can be moved at a time, the one at the top of a stack (not covered by another disk).
2. A disk must never be placed on a smaller one.

At the beginning, all disks are stacked on the left tower or peg, and at the end, they should be on the right tower, as illustrated in the following figure.

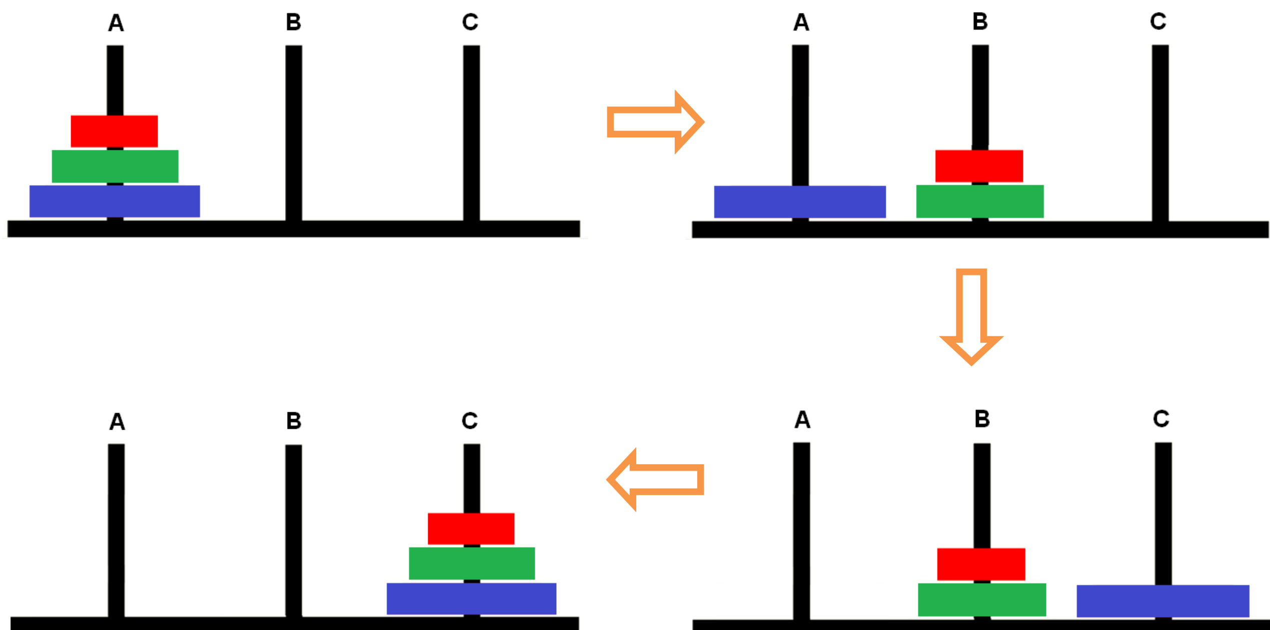


The problem may seem challenging, but the solution is easily expressed through recursion.

Let **n** be the number of disks to transfer. Suppose we know how to transfer **n-1** disks and want to transfer **n** from peg **A** to peg **C**, using peg **B** as an intermediate. We simply need to transfer the upper **n-1** disks from peg **A** to peg **B**, then transfer the last disk (the largest one) from **A** to **C** (which is free), and finally transfer the **n-1** disks from **B** to **C**, using peg **A** as an intermediate. The largest disk remains in place, and the process continues.

Example:

With n=3,



Remarks:

In fact, moving all disks except the largest one from one peg to another is itself a Tower of Hanoi problem with one less disk. It can, therefore, be solved in the same

manner described earlier. This process repeats until reaching the base case, which is moving a single disk. This is a recursive problem.

7.6.2) Solution

If we denote **`Hanoi(n, A, B, C)`** as the transfer of **`n`** disks from tower **`A`** to tower **`C`** using **`B`** as an intermediate, it can be achieved by:

- **`Hanoi(n-1,A,C,B)`**
- **`Hanoi(1,A,B,C)`** (this transfer actually does not use **`B`**)
- **`Hanoi(n-1,B,A,C)`**

The procedure to move **`n`** disks from **`A`** to **`C`** and display the steps is as follows:

```
Procedure Hanoi(n: integer; A, B, C: character);
Begin
If n = 1 Then Write("Move from ", A, " to ", C)
Else  Begin
      Hanoi(n-1, A, C, B);
      Hanoi(1, A, B, C);
      Hanoi(n-1, B, A, C);
      End;
End;
```

Example of Call:

Assuming the pegs are named left (**`L`**), middle (**`M`**), and right (**`R`**), the call would be:

`Hanoi(3,'L','M','R');`

7.7) Types of Recursion

7.7.1) Simple Recursion

Simple recursion occurs when the subprogram calls itself at most once, without calling any other recursive subprogram.

Example:

The previous function calculating the factorial of a number (**`Fact`**) exhibits simple recursion.

7.7.2) Multiple Recursion

Multiple recursion is when the subprogram calls itself more than once without invoking any other recursive subprogram.

Example:

The procedure that solves the Towers of Hanoi problem is a multiple recursive procedure.

7.7.3) Mutual Recursion (Cross Recursion)

We talk about mutual or cross recursion when a subprogram does not call itself directly but calls another subprogram which, in turn, calls the first one. Thus, cross recursion can be simple or multiple.

Example:

The parity test for an integer can be recursively formulated as follows:

- 0 is even and not odd.
- A number **`n`** is even if **`n-1`** is odd.
- A number **`n`** is odd if **`n-1`** is even.

7.7.4) Nested Recursion

Nested recursion involves making a recursive call within another recursive call.

***Example*:**

The Ackermann function (for two natural numbers, *m* and *n*) is defined as follows:

$$\mathrm{Ack}(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ \mathrm{Ack}(m-1,1) & \text{if } n = 0 \text{ and } m > 0 \\ \mathrm{Ack}\big(m-1, \mathrm{Ack}(m, n-1)\big) & \text{otherwise} \end{cases}$$

This definition exhibits nested recursion.