Algorithmic and Data Structure 2

Recursion

Definition

Recursion consists of **replacing a loop with a call to the sub-algorithm itself.**

- We call recursive any function or procedure that **calls itself.**

Illustrative Example 1: Factorial

- The best-known example is that of calculating the factorial defined by the formula:

N! = 1 * 2 * … * (N-2) * (N-1) * N, With 1! = 1 & 0! = 1

- **Iterative solution (using loop instruction)**

```
Fact ← 1;
 For i = 1 a N Do
        Fact ← Fact * i;
  Endfor ;
```

Illustrative Example 1: Factorial

- But another way we can say **that N! = N * (N-1)!**
- That is to say, the factorial of a number is the **result of multiplying the number by the factorial of the previous number.**
- If we want to program this calculation we can imagine **a Fact function** which performs the multiplication of the number passed as an argument by the factorial of the previous number, and this factorial of the previous number will of course itself be calculated by the same Fact function in another call.

Illustrative Example 1: Factorial

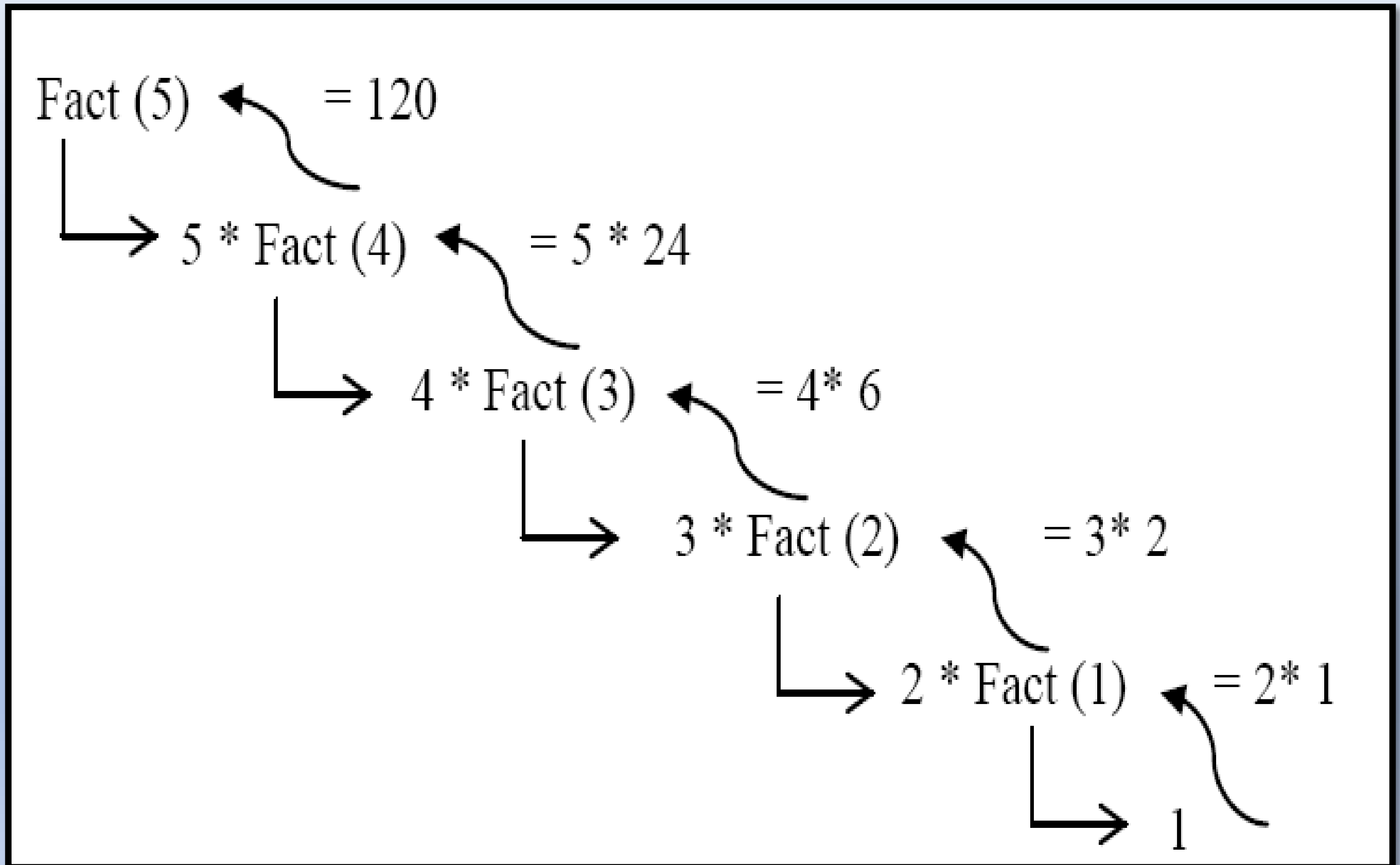
- **Recursive solution:** The recursive function of the factorial is written as follows:

```
Function Fact (N: Integer): Integer;
Begin
    If N = 1 Then
            Fact ← 1;
    Else
     Fact ← N * Fact (N-1);
    Endif
End;
```

Flow of the Recursive function

- The execution of the recursive function therefore amounts to making **repetitive calls** to the function until **the stopping condition** is validated.
- Let's take the recursive function of the factorial and run it **for N = 5;**
- In the diagram on the next slide, the downward arrows represent the calls (from Fact (5) to Fact (1)) we then arrive at the execution of fact (1) for which the stopping condition is valid then the return is done backwards (in the opposite direction) and each call then corresponds to the calculated value.

Flow of the recursive function



Stop condition of a recursive function

- Since a recursive function calls itself, **it is imperative that we provide a stopping condition otherwise the program will never stop!**
- We must always test the stopping condition first, and then, if the condition is not verified, then make a recursive call.
- In the case of the factorial function above the stopping test can also be when **N = 0** because by definition 0! is known and equal to 1.

How to write a recursive algorithm/program

To write a recursive algorithm you must analyze the problem to:

- ✓identify the **particular case (s) (stopping condition)**.
- ✓identify the general case that performs the recursion.

Example 2: Fibonacci sequence

- **Fibonacci** is a sequence of integers in which each term is the sum of the two terms that precede it:

  $U_0 = 1, \; U_1 = 1$

  $U_n = U_{n-1} + U_{n-2}$ for $n > 1$

```
Function Fibo (N: Integer): Integer;
Begin
  If (N = 0) or (N = 1) Then
     Fibo ← 1;
 Else
      Fibo ← Fibo (N-1) + Fibo (N-2);
Endif
End;
```

Error handling

- In the two previous functions (**factorial** and **fibonacci**) the value of **N** must be a **positive integer** otherwise we fall back on the case of the infinite loop.
- Assuming that we want to calculate the factorial of (-1).
- In the case of the iterative solution (the loop "For i<- 1 a N Do") the loop will never be triggered because N being less than 1 (the minimum value) but the execution will give a false result, "1" which the Fact result was initialized.

Error handling

- In the case of the recursive function N =-1 not being equal to 0, execution of the instruction " Fact<- N * Fact (N-1)" call the function Fact (-2) which will subsequently trigger a call to Fact (-3)…. without ever arriving at the stopping test and therefore we find ourselves in the case prohibited in any algorithm, namely **INFINITY** .
- It is therefore necessary to call Fact only for all N whose value is greater than 1 (otherwise greater than 0 if we modify the function stopping test to "If N = 0 then".

Error handling

- A value test before the Fibonacci call is therefore necessary (N must have at least the value 0) in order to avoid the infinite case in the function.

- ***Note: It is therefore important to specify that the treatment of error cases (unaccepted values) must be managed in the calling algorithm (or sub-algorithm) in order to allow correct execution of the recursive sub-algorithm.***

Example 3: Sum of the first N positive natural numbers

- The mathematical formula for this calculation is $S = \sum_{n}^{1} i$ in other words

S = 1+2+3+…. +N or again:

$U1 = 1$

$U_n = U_{n-1} + N$ with $N \geq 1$

- **Sequential function (iterative solution)**

```
Function Suite (N: integer): integer;
Variables i, S: Integer;
Begin
   S <- 0;
For i <- 1 to N do
     S <- S + i;
EndFor
Suite <- S;
End;
```

Example 3: Sum of the first N positive natural numbers

- Writing the same function recursive gives the following code:

```
function Suite-R (N: Integer): integer;
Begin
If N = 1 Then
    Suite-R ← 1 ;
else
Suite-R ← Suite-R (N-1) + N ;
Endif
End;
```

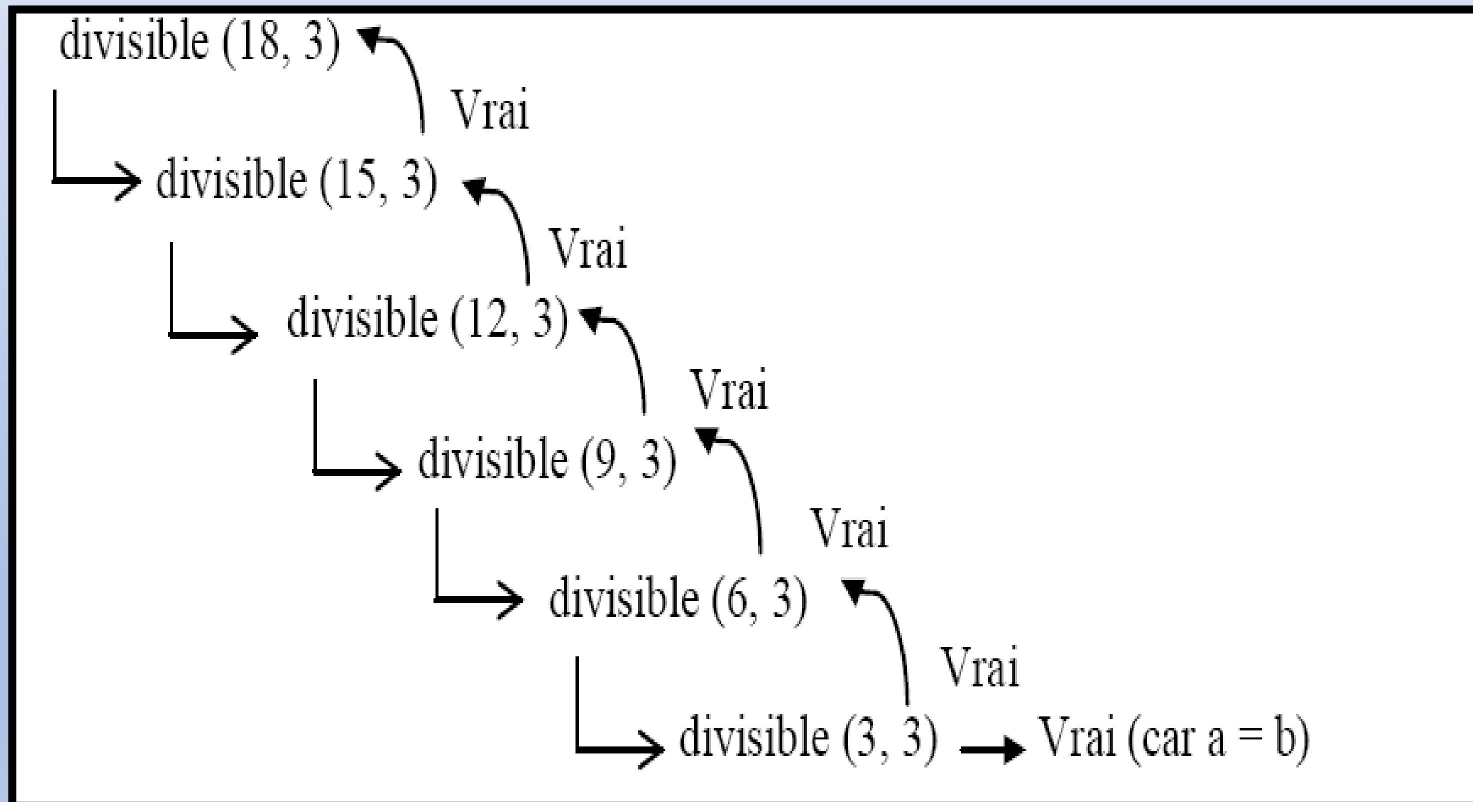
Example 4: Number divisible by another

- This Boolean function returns a value ***true*** if a number **"b" is a divisor of a number "a"** or false otherwise.

```
Function Divisible (a, b: integer): boolean ;
Begin
If (a =b) then
    divisible ← true;
else if (a<b) then
        divisible ← false;
    else
        divisible ← divisible (a –b, b);
Endif
End;
```

```
bool divisible (int a, int b){
if (a==b){
    return true; }/*true*/
else if (a<b){
            return false;}
else{
    return (divisible(a-b,b));}
}
```

Example 4: Number divisible by another



Example of recursion in C

/*This program calls the power recursive function to calculate x^y (x integer and y positive integer*/

```
#include < stdio.h >
int power ( int , unsigned ); /*Declaration (prototype) of the power function*/
main( ) {
int x;
unsigned y; /*y positive integer*/
scanf ("%d%u",&x,&y); / *reads the values of x and y*/
printf ("%d",power( x,y )); /*displays the result of x pow y */ }
int power ( int x, unsigned y) /*definition of the power function*/
{
if (y==0)
     return(1); /*returns 1 if y=0*/
 else /*returns x*xy-1 (calls power again for x and y-1)*/
    return ( x * power (x,y-1) );
}
```

Types of recursion

- We distinguish four types of recursion:
  - Nested Recursion,
  - Cross Recursion,
  - Terminal Recursion,
  - Non-terminal Recursion.

Nested recursion

- ***Principle*** : This type of recursion consists of making a recursive call inside another recursive call.
- The most used example to present this type of recursion is that of the **Ackerman sequence** defined by :

A (m,n) = n+1 if m = 0,

A (m,n) = A(m-1,1) if n=0 and m > 0

A (m,n) = A(m-1, A(m,n-1)) otherwise

Nested recursion

- A recursive function to calculate a term A (m, n) looks like this:

```
Function Ackerman (m, n: Integer): integer;
Begin
If m = 0 then
    Ackerman ← n +1
else
If (n = 0) and (m > 0) then
        Ackerman ← Ackerman (m-1, 1)
else
 Ackerman ← Ackerman (m-1, Ackerman (m, n-1))
   Endif
Endif
End ;
```

Cross recursion

- ***Principle*** **:** consists of writing functions that call each other. In this case, the order of the two functions is not important, however it is necessary to:

1- Write the header of the second function (the first function can call it);

2- Write the first function in full;

3- Write the second function in full;

Cross recursion

- Consider the following calculation:

  A(x) = 1 if $x \leq 1$

  A(x) = B (x+2) if $x > 1$

  With: B(x) = A(x-3) +4

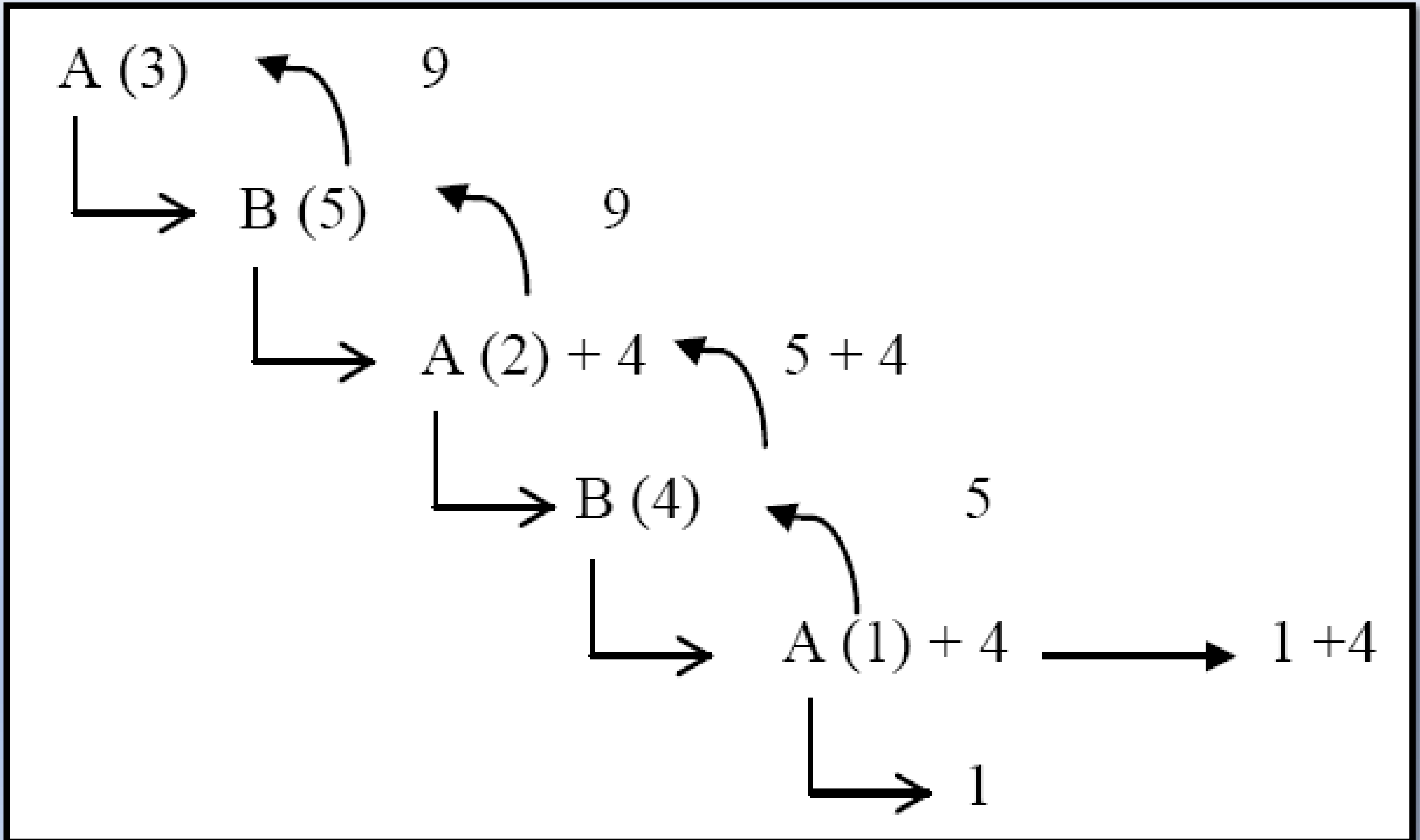
- **Function B (x: integer): Integer; {header of function B}**
- **Function A (x: integer): Integer;**

```
Begin
If x ≤ 1 then
   A ← 1
else
   A ← B (x+2)
 Endif
 End;
```

```
Function B (x: integer): Integer;
Begin
B ← A (x-3) +4;
End;
```

Cross recursion

- Execution for A (3)



Terminal and non-terminal recursion

- ***Terminal recursive*** **:** A sub-algorithm is said to be terminal if no processing is carried out on the return of a recursive call except the return of a value (See the example of a number divisible by another).

- ***Non-terminal recursion*** **:** where the result of the recursive call is used to carry out processing; in addition to the return of a value, the calculations are done on the return (see the Factorial function)

Conclusion

- Recursive algorithms are simple (it's just another way of thinking).
- Recursive algorithms are used to solve complex problems.